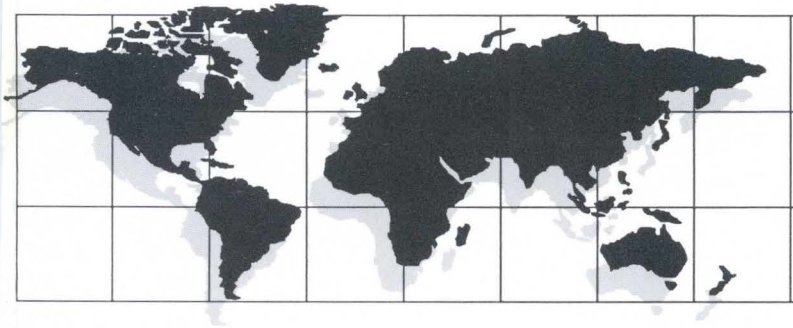


CTOS

Programming Guide

Volume 1

General Programming Topics

UNISYS

UNISYS

CTOS®

Programming Guide
Volume 1
General Programming Topics

Copyright © 1990, 1991, 1992, 1993 Unisys Corporation
All Rights Reserved
Unisys is a registered trademark of Unisys Corporation

CTOS III 1.1
CTOS II 3.4
CTOS I 3.4
CTOS/XE 3.4.100
Development Utilities 12.3
Standard Software 12.3
Video Access Method 4.2

November 1993

Printed in USA
4586 7900-000

The names, places, and/or events used in this publication are not intended to correspond to any individual, group, or association existing, living, or otherwise. Any similarity or likeness of the names, places, and/or events with the names of any individual, living or otherwise, or that of any group or association is purely coincidental and unintentional.

NO WARRANTIES OF ANY NATURE ARE EXTENDED BY THIS DOCUMENT. Any product or related information described herein is only furnished pursuant and subject to the terms and conditions of duly executed agreement to purchase or lease equipment or to lease software. The only warranties made by Unisys, if any, with respect to the products described in this document are set forth in such agreement. Unisys cannot accept any financial or other responsibility that may be the result of your use of the information in this document or software material, including direct, special or consequential damages.

You should be careful to ensure that the use of this information and/or software material complies with the laws, rules, and regulations of the jurisdictions with respect to which it is used.

The information contained herein is subject to change without notice. Revisions may be issued to advise of such changes and/or additions.

RESTRICTED RIGHTS LEGEND. Use, reproduction, or disclosure is subject to the restrictions set forth in DFARS 252.227-7013 and FAR 52.227-14 for commercial computer software.

Copyright © 1993 Unisys Corporation
All Rights Reserved
Unisys is a registered trademark of Unisys Corporation

Convergent, Convergent Technologies, CTOS, NGEN, and SuperGen are registered trademarks of Convergent Technologies, Inc.

ClusterCard, Context Manager, CT-Net, CTOS/VM, CTOS/Vpc, Document Designer, Generic Print System, PC Emulator, Print Manager, shared resource processor, SRP, Voice/Data Services, Voice Processor, X-Bus, and X-Bus+ are trademarks of Convergent Technologies, Inc.

OFIS is a registered trademark of Unisys Corporation.

BTOS is a trademark of Unisys Corporation.

OS/2 is a registered trademarks of International Business Machines Corporation. Microsoft is a registered trademark of Microsoft Corporation. Presentation Manager and Windows are trademarks of Microsoft Corporation.

UNISYS

Product Information Announcement

● New Release ○ Revision ○ Update ○ New Mail Code

Title

CTOS® Programming Guide, Volume 1, General Programming Topics

This Product Information Announcement announces the release and availability of the CTOS Programming Guide, Volume 1, General Programming Topics.

This two-volume set describes general programming techniques and the extended system services provided with the CTOS operating systems. It is meant to be used in conjunction with *CTOS/Open Programming Practices and Standards.* This new edition includes information on programming Standard VGA, writing dynamic link libraries under CTOS, and writing system services for multiprocessor EISA-bus workstations.

Please address all technical communications relative to this document to
UNISYS, Multimedia Publishing M/S 18-007, 2700 North First Street,
San Jose, CA 95134-2028.

To order additional copies of this document

- United States customers, call Unisys Direct at 1-800-448-1424.
- All other customers, contact your Unisys Sales Office.
- Unisys personnel, use the Electronic Literature Ordering (ELO) system.

Contact your Unisys representative for pricing information.

4586 7900-000 Volume 1 only
4586 7918-000 Volume 2 only
4357 9515-100 Volumes 1 and 2

CTOS is a registered trademark of Convergent Technologies, Inc.

Announcement only:
SA, SN, SU, SW, PR5

Announcement and attachments:

System: CTOS
Release:
CTOS III 1.1
CTOS II 3.4
CTOS I 3.4
CTOS/XE 3.4.100
Development Utilities 12.3
Standard Software 12.3
Video Access Method 4.2

Part Number: 4586 7900-000

Contents

Volume 1

About This Guide

Introduction xix
What This Guide Covers xix
Who Should Use This Guide xx
How This Guide Is Organized xx
What Is New in This Guide xxii
Where to Find More Information xxiv

Section 1. The CTOS Programming Environment

Introduction to CTOS Programming 1-1
Distributed Client-Server Computing: The CTOS Cluster 1-1
Tools Available to the CTOS Programmer 1-2
CTOS Development Utilities 1-2
Language Compilers and Interpreters 1-2
Additional Programmer's Libraries 1-3
An Overview of CTOS System Software 1-3
General Steps in Writing a CTOS Program 1-6
Writing Source Code 1-6
Compiling a CTOS Program 1-7
Linking a CTOS Program 1-7
Running the Program 1-8
Naming Conventions 1-9
Data Types 1-9
Defining the Basic Data Types 1-10
Variable Names 1-10
Prefixes 1-10
Roots 1-12
Suffixes 1-13
Examples of Variable Names 1-13
Procedure Naming 1-14
File Suffix Conventions 1-14
Models of Segmentation 1-17
Your Compiler and Models of Segmentation 1-17
Procedural Calls Between Programming Languages 1-18
Your Compiler and Calls Between Languages 1-18
Some Configuration Tips for Programmers 1-19
The CTOS Debugger 1-19
Using Consistent Library Versions 1-20

Section 2 Introduction to Protected Mode

Introduction to Protected Mode 2-1
Advantages of Protected Mode 2-2
Review of Segmented Addressing 2-2
Review of Real-Mode Addressing 2-3
Protected Mode Addressing 2-4
Memory Mapping Using 80386 Paging 2-6
The Format of a Selector 2-7
The Segment Descriptor Tables 2-7
Local Descriptor Table (LDT) 2-7
Global Descriptor Table (GDT) 2-8
The Contents of a Segment Descriptor 2-9
Processor Exceptions and Faults 2-11
Gate Descriptors 2-12
Protection Models 2-12

Section 3 Using Color

Introduction to Color 3-1
Video Concepts 3-2
Program Interfaces 3-4
ProgramColorMapper 3-5
ProgramColorMapper Structures 3-6
Single-Palette Format 3-7
Advantages Over Three-Palette Format 3-9
Three-Palette Format 3-9
Color Structure 3-11
Field Descriptions 3-12
Alphanumeric, Background, and Graphics 3-14
Enabling Background Color Capability 3-14
Effect of fBackgroundColor on Palettes 3-15
Background Color and Three-Palette Format 3-15
Background Color and Single-Palette Format 3-16
More Examples in Three-Palette Format 3-18
Case 1: Using Alphanumeric Control Structure Fields . 3-18
Case 1: Condition A 3-18
Case 1: Condition B 3-19
Case 1: Condition C 3-20
Case 1: Condition D 3-20
Case 2: Using Graphics Control Structure Fields 3-21
Case 2: Condition A 3-21
Case 2: Condition B 3-22
Case 3: Using Alpha and Graphics Fields 3-22

Maximizing Use of a 16-Entry Palette 3-23
Gray-Scale Monitors 3-23
Color Application Notes 3-24
Combining Alphanumeric Color With the Values in the Attribute Byte 3-24
Defining Bitmap Color 3-27
Color Priorities 3-28
Color Programming Tips 3-30
Avoid Use of Reverse Video and Graphics 3-30
Querying Single-Palette Format 3-30
Avoid Combining Alpha Background with Graphics 3-31
Color Program Examples 3-31
Three-Palette Format Example 3-32
Single-Palette Format Example 3-40

Section 4 Converting CTOS VGA to Standard VGA

Overview 4-1
Who Should Use This Information 4-2
General Concepts and Terminology 4-2
VGA Font Capabilities 4-6
Nine-Bit-Wide Fonts 4-6
Standard VGA Capability 4-7
CTOS VGA Capability 4-8
Problem Description 4-10
Affected Applications 4-10
Recommended Solution 4-10
Half-Bit Shift 4-12
Standard VGA Capability 4-12
CTOS VGA Capability 4-12
Problem Description 4-13
Affected Applications 4-13
Recommended Solution 4-13
Display Resolution 4-15
Standard VGA Capability 4-15
CTOS VGA Capability 4-16
Problem Description 4-16
Affected Applications 4-17
Recommended Solution 4-17

Character Attributes 4-17
Standard VGA Capability 4-17
CTOS VGA Capability 4-20
Problem Description 4-21
Affected Applications 4-22
Attributes 4-22
Color 4-22
Character Map 4-22
Recommended Solution 4-23
Mapping Attributes from CTOS to Standard VGA 4-23
Default Attribute Mapping 4-24
Default Attribute Mapping Tables 4-29
Accessing the Character Map 4-40

Section 5 Writing Partition-Managing Programs

Introduction to Partition Management 5-1
A Review of Partition Management Operations 5-1
Creating a Partition and Loading a Program Into It 5-2
Creating a Partition 5-2
Swapping the Partition Context Into Memory 5-3
Setting Up the Program's Environment 5-4
Loading the Program into the New Partition 5-6
Finding the Program's Termination Status 5-6
The Child-Termination Question 5-6
A Solution: Defining a Termination Request 5-7
Calling the Termination Procedure 5-7
Structure of the Termination Procedure 5-8
Deallocating the Partition 5-9
A Sample Partition-Management Program 5-10

Section 6 Software Installation: The Installation Manager

Introduction to Installation Manager 6-1
Key Concepts 6-2
Types of Installation: Floppy, Tape, Server 6-2
Public vs. Private Installation 6-3
Subpackages 6-3
The Installation Manager 6-4
Batch 6-6
Installation Files 6-6
Control File 6-7
Control File Keywords 6-8
Installation Script File 6-13
Message File 6-13
Command File 6-14
User Configuration File 6-15
Naming Your Floppy Installation Files 6-16
Naming Your Tape Installation Files 6-18
Organizing Your Installation Media 6-20
For a Floppy Installation 6-20
For a Tape Installation 6-23
Installation Variables 6-23
File Lists 6-27
Restarting an Installation 6-28
Nationalization 6-29
Installation Tips 6-30
Installation Manager Version 12.0 and Earlier 6-30
Using Variables 6-31
Messages 6-32
Packages and Subpackages 6-32
Volumes and Directories 6-32
File Lists 6-33
Other Tips 6-33
Installation Script Examples 6-34
Example 1: One Subpackage 6-35
Install.ctrl File 6-35
InstallMsg.bin File 6-35
Install.cmds File 6-35
Install.jcl File 6-36

Example 2: Nested Subpackages 6-41
Install.ctrl File 6-42
DevelopmentUtilities>Install.ctrl File 6-42
DevelopmentRunFiles>Install.ctrl File 6-43
DevelopmentRunFiles>InstallMsg.bin File 6-43
DevelopmentRunFiles>Install.cmds File 6-43
DevelopmentRunFiles>Install.jcl File 6-44
DevelopmentLibraries>Install.ctrl File 6-49
DevelopmentLibraries>InstallMsg.bin File 6-50
DevelopmentLibraries>Install.jcl File 6-50
AsynchronousExamples>Install.ctrl File 6-55
AsynchronousExamples>InstallMsg.bin File 6-55
AsynchronousExamples>Install.cmds File 6-55
AsynchronousExamples>Install.jcl File 6-56
Example 3: Tape Installation 6-63
InstallMsg.bin File 6-64
Install.jcl File 6-64

Section 7 Using the System Log File

Introduction to System Log File 7-1
Log File Format 7-2
Writing Messages to the Log File 7-2
Displaying Messages Using PLog 7-3
Log File Fields in the Volume Home Block 7-3
How the File System Writes to the Log File 7-4
Record Fits in the Buffer 7-4
Record Does Not Fit in the Buffer 7-4
How to Read the Log File 7-6
Accessing the Log-File Fields in the VHB 7-6
Determining if Records Have Wrapped Around 7-6
Reading the File in Chronological Order 7-7
If Records Have Not Wrapped Around 7-7
If Records Have Wrapped Around 7-8
Reading the File in Reverse Chronological Order 7-8
Saving Offsets to Records 7-8
If Records Have Not Wrapped Around 7-9
If Records Have Wrapped Around 7-9
PLog's Algorithm For Processing Each Sector ... 7-10

Section 8 Writing Multiprocessor System Services

Introduction 8-1
Portation Issues for Existing Programs 8-1
General Guidelines 8-2
Unsupported Mechanisms 8-2
User Numbers and Exchanges 8-2
Remote Memory and Inter-CPU Communication 8-3
Restricted Use of GetWSUserName on SRPs 8-3
Controlling Interprocessor Request Routing 8-3
Interprocessor Request Routing Types 8-4
Routing by Device Specification 8-6
Deinstalling an Instance of a System Service 8-14
Remote Routing 8-15
Local Routing 8-18
Other Routing Methods 8-19
Enhancing Remote Data Transfer Requests on SRPs .. 8-19
Use of Handles on Multiprocessor Systems 8-20
The Standard Connection Handle 8-21
Non-Standard Handle Types 8-21

Section 9 The Synchronous CommLine Interface

Introduction to CommLine Interface 9-1
The CommLine Interface 9-1
Extensions to the Traditional CommLine Interface 9-1
InitCommLine 9-2
Differences Between 8274 and 8530 Communication Controllers 9-3
ChangeCommLineBaudRate 9-4
ReadCommLineStatus 9-4
WriteCommLineStatus 9-5
Using DMA with Synchronous Data Communication 9-5
Initializing Communications DMA 9-6
TransmitCommLineDMA 9-7
ReceiveCommLineDMA 9-7
GetCommLineDMAStatus 9-8
Implementing X.21 (1984) Protocol Support 9-9
What Is the X.21 Protocol? 9-9
Features of the X.21 Support Hardware 9-10
Initializing a Communications Line with X.21 Support Enabled 9-12
Using the X.21 Support Hardware in Drivers-Only Mode 9-12
Using V.35 Support Hardware 9-13

Section 10 Creating a Dynamic Link Library

Introduction 10-1
Assumptions 10-1
Supporting Documentation 10-2
Software and Tools 10-2
Introducing the DLL Example 10-4
Queue DLL 10-4
DLL Source Files 10-5
Overview of Queue.c 10-6
Queue.c Declarations 10-6
Global Data Declarations 10-6
Instance Data Declarations 10-6
Queue Pointer Procedures 10-7
IncrementQueuePtr 10-7
QueueUsedSpace 10-8
QueueFreeSpace 10-8
ReadQueueIntoMemory 10-9
WriteQueueFromMemory 10-9
Initialization and Termination Procedures 10-9
Export Procedures 10-10
Mediating Medium-Model Library Procedures 10-10
Setting Up the Initialization Procedure 10-11
Using a Termination Procedure 10-12
Using Semaphores to Handle Global Data 10-14
In QueueInitialization 10-14
In QueueTermination 10-15
In The Export Procedures 10-16
Setting Up Global Data Variables 10-17
Setting Up Instance Data Variables 10-18

Preparing the DLL 10-19
Overview of Preparing a DLL 10-20
Makefile and Executive Commands 10-22
Compiling Queue.c 10-23
Creating the Object Module and Import Library .. 10-24
Linking The DLL Object and Library Files 10-25
Using the DLL 10-27
Client Source Files 10-27
Reading Queue Entries 10-28
Writing Queue Entries 10-28
Preparing the Client Source Files 10-29
Client Makefile and Executive Commands 10-31
Compiling ReadQueue.c 10-31
Linking the Client Object and Library Files 10-32
Adding Resources 10-34
DLL Program Listings 10-35

Volume 2

Section 11 Mouse Services

What Are the Mouse Services? 11-1
Mouse System Service 11-1
Object Module Library 11-1
Functional Groups 11-2
Examples 11-3
Important Concepts 11-3
Using the Mouse Buttons 11-4
Screen Coordinates 11-4
Normalized Screen Coordinates 11-4
Virtual Screen Coordinates 11-4
Setup 11-6
Queries 11-7
Cursor Shape 11-12
Changing the Graphics Cursor 11-14
Cursor Movement 11-15
Troubleshooting 11-16
Sample Mouse Program 11-17

Section 12 Queue Manager

Introduction to Queue Manager 12-1
What Is the Queue Manager? 12-1
Run Files 12-2
Installation and Deinstallation 12-2
Functional Groups of Queue Management Operations ... 12-3
Queue Manipulation Operations 12-3
Queue Server Operations 12-4
Establishing Queue Servers 12-4
Marking Queue Entries 12-4
Unmarking Queue Entries 12-5
Processing Queue Entries 12-5
Removing and Rescheduling Queue Entries 12-5
Rewriting Queue Entries 12-5
Client Operations 12-6
Adding an Entry to a Queue 12-6
Deleting an Entry 12-7
Reading Queue Entries 12-7
Summary of Queue Manager Operations 12-7
Queue Manager Configuration 12-10
Queues 12-11
Format 12-11
Queue File Header 12-11
Queue Entry 12-12
Queue Entry Processing Order 12-12
Queue Entry Format 12-13
Calculating Queue Entry Size 12-13
Queue Examples 12-14
Defining Queues 12-14
Defining Queues in the Queue Index File 12-14
Defining Queues Dynamically 12-16
Referencing Queues and Queue Entries 12-17
Referencing Queues 12-17
Queue Names 12-17
Queue Handles 12-18
Referencing Queue Entries 12-18
Queue Entry Handles 12-18
Queue Status Block 12-19
Keys 12-19
Sequence for Using Queue Management Operations 12-21
Using the Queue Manager Across Network Nodes 12-22
Data Structures 12-24

Section 13 Spooler

Introduction to Spooler 13-1
Spooler Configuration 13-1
Sending a Password 13-2
Operations 13-2
Programmer's Notes on the Spooler 13-3
Pre-GPS Spooler Byte Streams 13-3
Spooler Configuration File Requirements 13-4
Printer Spooler Escape Sequences 13-5
Queue Management 13-5
Scheduling Queue 13-6
Status Queue 13-6
Control Queue 13-6
ConfigureSpooler Program Example 13-7
SpoolerPassword Program Example 13-7
Data Structures 13-8

Section 14 Voice/Data Services

Overview 14-1
Telephone Management 14-2
Telephone Status Monitor Program 14-2
Data Management 14-2
Functional Groups of Operations 14-2
Voice Management Operations 14-3
Telephone Management Operations 14-3
Data Management Operations 14-5
Hardware Features of the X-Bus Workstation 14-6
Voice Features 14-6
CODEC 14-6
Voice Amplifier 14-6
Telephone Features 14-6
Two Telephone Lines 14-7
DTMF Tone Generator and Receiver 14-7
Call Progress Tone Detector (CPTD) 14-9
Data Features 14-9
Modem 14-9
Analog Crosspoint Switch Array 14-9

Software Concepts 14-10
Voice Recording 14-10
Recording Rates 14-10
Pause Compression 14-10
Amplification 14-11
Memory and Disk Files 14-12
Memory Usage 14-12
Structure of a Voice File 14-13
Adaptive Differential Pulse Code Modulation (ADPCM) 14-13
Voice Control Structure 14-14
Typical Sequence for Record/Playback 14-14
Multiple Voice Messages in One File 14-14
Voice Playback from Memory 14-15
Telephony 14-15
Voice and Data Lines 14-15
Telephone Unit vs. Telephone Line 14-16
Hold 14-16
Dialing 14-16
Generating DTMF Tones 14-18
Internationalized Call Progress Tone Detection .. 14-18
Sample Program Using Internationalized Call Progress-Tone Detection System 14-21
Data 14-22
Starting a Data Call 14-23
Converting a Voice Call to a Data Call 14-23
Accepting a Data Call 14-23
Originating a Data Call 14-24
Reading and Writing Data 14-24
Terminating a Data Call 14-24
Telephone Status Debugging Tool 14-25
Command Form 14-25
Parameter Field 14-25
Operation 14-26
Lines J0 through J3 14-27
Lines L0 through L7 14-28
Status Monitor Function Keys 14-29

Program Examples 14-30
Example 14-2: Dialing 14-30
Example 14-3: Voice Response System 14-35
Setting Up Request Blocks 14-35
fNoWait Flag 14-35
Use of TsVoiceConnect 14-36
Possible Modifications to This Program 14-36
Example 14-4: Voice Memory Playback 14-57
Example 14-5: Data Call 14-69
Data Structures 14-81

Section 15 Performance Statistics Service

Overview 15-1
Functional Groups of Operations 15-2
Statistics Session Operations 15-2
Logging Session Operations 15-2
General-Purpose Operations 15-2
Statistics Session 15-3
Statistics ID Block 15-3
Block Number 15-3
Index 15-3
Opening a Statistics Session (PsOpenStatSession) ... 15-4
Block ID 15-4
Session Handle 15-5
Disk Activity 15-5
Getting the Counters (PsGetCounters) 15-6
Closing a Statistics Session (PsCloseSession) 15-8
Logging Session 15-8
Opening a Logging Session (PsOpenLogSession) 15-8
Reading a Log (PsReadLog) 15-9
Closing a Logging Session (PsCloseSession) 15-10
Program Example 15-10
Data Structure 15-17

Section 16 Asynchronous System Service Model

Introduction to Asynchronous Services 16-1
Terminology 16-1
Synchronous and Asynchronous System Service Models 16-3
Writing an Asynchronous System Service 16-6

Async.lib Procedures 16-10
Async.lib Procedures You Can Use in the Main Module .. 16-10
Requesting a Service on Behalf of a Client 16-10
Review of the Synchronous Request Procedural Interface 16-10
Asynchronous Request Procedural Interface 16-11
Passing a Variable Length Parameter List in PL/M 16-12
Building Request Blocks 16-14
AsyncRequest 16-14
AsyncRequestDirect 16-14
Checking the Context Stack 16-15
CheckContextStack 16-15
Heap 16-15
Allocating and Deallocating Heap Memory 16-16
Conserving Heap Memory 16-17
System Requests 16-17
Terminating and Aborting Requests 16-17
Swapping Requests 16-18
Handling System Requests 16-19
Handling Termination and Abort Requests 16-19
Handling Swapping Requests 16-21
Debugging Aids 16-22
Examining the Logging Module LogAsync 16-22
Maintaining Debugging Statistics 16-24
Async.lib Procedures Used by the Common-Code Module 16-25
Managing Contexts 16-25
Creating a Context 16-26
Context Control Block 16-27
Resuming a Context 16-27
Terminating a Context 16-28
Other Ways Contexts Can Be Used 16-28
Terminating Contexts at Deinstallation 16-29
Using the Heap 16-29
Managing the Heap 16-30
Logging Messages for Debugging Purposes 16-32
Initializing 16-32
Availability of Asynchronous System Service Files 16-33
Linking Your System Service 16-34
Program Example 16-36
AsyncService.c 16-36
Example.c 16-37
Start.c, Stop.c, and DeInstall.c 16-37

Section 17 CD-ROM Service

CD-ROM Overview 17-1
Requirements 17-1
Functional Groups of Operations 17-2
Volume Information 17-2
Status 17-3
Read File 17-3
Audio 17-3
Miscellaneous (Rarely Used) 17-4
Standard File Formats 17-4
Determining the File Format Using CdGetVolumeInfo .. 17-4
Other Uses of CdGetVolumeInfo 17-6
File Structure: Hierarchical vs. Flat 17-6
CTOS File Specification (for Flat File Structures)..... 17-7
Backslash File Specification (for Hierarchical File Structures) 17-8
Obtaining the Directory List 17-9
Example: Using CdDirectoryList 17-10
Searching for Files 17-13
Example: File Search 17-14
Copying a CD-ROM File to Disk 17-19
Example: Copying to Disk 17-19
Using Audio Features of CD-ROM 17-24
Specifying Locations on the CD-ROM Disc 17-24
Q-Channel 17-24
Audio Example 17-25
File Formats 17-33
Primary Volume Descriptor Formats 17-33
ISO Directory Record Format 17-49
Character Sets 17-58
d-characters 17-58
a-characters 17-59
c-characters 17-60
a1-characters 17-60
d1-characters 17-60
Separators 17-60

Section 18 Sequential Access Service

Overview of Sequential Access Service 18-1
Functional Groups of Operations 18-2
Basic Operations 18-2
Advanced Operations 18-3
Miscellaneous 18-3
General Model of Sequential Access Devices 18-4
Data Storage Characteristics 18-4
Logical Elements within a Tape 18-6
Logical Data Blocks 18-6
Interblock Gaps 18-7
Filemarks 18-7
Blank Space 18-7
Erase Gaps 18-7
Data Buffering 18-8
Device Buffers 18-8
Sequential Access Service Buffers 18-9
Allocating Memory for Recovering Buffer Data ... 18-9
Residual Data 18-9
Buffer Recovery Order 18-12
Example 18-13
SeqAccessCheckpoint 18-14
Specifying Buffer Sizes 18-16
Fixed-Length and Variable-Length Records 18-16
Increasing Block Size: Pros and Cons 18-17
Programming Considerations 18-17
Record Size and Block Size 18-17
Recording Density and Transport Speed 18-18
Buffered Mode 18-19
Erase to EOM After Close 18-19
Suppress Default Mode on Open 18-19
Buffer Recovery Order 18-19
Examples 18-20
Example 1: Fixed-Length, Blocked Records 18-20
Example 2: Variable-Length Records 18-24

Figures

1-1. Where CTOS System Calls Are Processed 1-5

2-1. Real Mode Segment Addressing 2-3
2-2. Protected Mode Segment Addressing 2-5
2-3. Protected Mode Address Translation for Paging 2-6
2-4. The Format of a Selector 2-7
2-5. Pointer Aliasing 2-10
2-6. Separate Address Spaces Protection Model 2-13

3-1. Character Cell with Foreground and Background Pixels 3-3
3-2. Example Palette in the Single-Palette Format 3-8
3-3. Three-Palette Format and Single-Palette Format 3-13
3-4. Sample Three-Palette Format 3-15
3-5. Sample Single-Palette Format 3-17
3-6. Case 1 Palette Settings 3-18
3-7. Case 2 Palette Settings 3-21
3-8. Sample Palette 3-28
3-9. Color Priorities 3-29

4-1. Character Cell 4-3
4-2. Scan Lines 4-4
4-3. Relationship of a Character Cell to a Screen Display 4-5
4-4. Characters Codes 0 through 127 4-7
4-5. Characters Codes 128 through 255 4-8
4-6. Representing 9-Bit-Wide Characters 4-9
4-7. Converting to an 8x12 Font 4-11
4-8. Half-Bit Shift 4-12
4-9. Effects of Ignoring Half-Bit Shift 4-14
4-10. Standard VGA: Option One 4-17
4-11. Standard VGA: Option Two 4-18
4-12. Standard VGA: Option Three 4-18
4-13. CTOS Attributes Without Color 4-20
4-14. CTOS Attributes With Alphanumeric Color 4-20
4-15. Standard VGA Default Attributes 4-24
4-16. Mapping CTOS Attributes 4-41

6-1 Files for Floppy Installation of Accounting Package 6-17
6-2 Files Used in a Tape Installation for an Accounting Package 6-19

7-1. Log File Wraparound 7-5

10-1 Queue DLL 10-4
10-2. IncrementQueuePtr 10-8
10-3. Leaving-One-Byte-Free 10-9
10-4. Preparing the DLL 10-21
10-5. Preparing the ReadQueue Client 10-30

11-1. Virtual and Normalized Screen Coordinates 11-5
11-2. Cursor Movement Inside and Outside a Motion Rectangle 11-9

12-1. Configuration with the Queue Management Facility 12-10
12-2. Example of a Queue Index File 12-16

14-1. Parts of the Telephone Unit 14-7
14-2. Voice Processor Module Connections 14-8
14-3. Telephone Status Command Screen 14-26
14-4. Telephone Status Screen with Telephone Offhook 14-27

16-1. Program Flow for the Synchronous Model 16-5
16-2. Program Flow for the Asynchronous Model 16-7
16-3. Source Modules to Run File 16-9

17-1. Flat File Structure 17-7
17-2. Hierarchical File Structure 17-8

18-1. General Layout of a Tape 18-5
18-2. Serpentine Recording (QIC Tape) 18-5
18-3. Parallel Recording (Half-inch Tape) 18-6
18-4. Helical Scan Recording (DDS) 18-6
18-5. Example of Variable-Length Records 18-18

Tables

1-1. Basic Data Types 1-9
1-2. Common Prefixes 1-11
1-3. Common Roots 1-12
1-4. Common File Suffixes 1-15

4-1. Display Resolutions Supported by CTOS Standard VGA VAM 4-15
4-2. CTOS VGA Display Resolutions 4-16
4-3. Standard VGA Default Colors 4-19
4-4. CTOS VGA Default Colors 4-21
4-5. CTOS Standard VGA VAM Default Colors 4-25
4-6. Standard VGA Attributes (0 - F) 4-26
4-7. Standard VGA Attributes (10 - 1F) 4-27
4-8. Standard VGA Attributes (x0 - xF) (where x = (2,3,4,5,6,7)) 4-28
4-9. Standard VGA Attributes (x0 - xF) (where x = (8,9,A,B,C,D,E,F)) ... 4-29
4-10. Monochrome Attribute Mapping (00 - 0F) 4-30
4-11. Monochrome Attribute Mapping (10 - 1F) 4-30
4-12. Monochrome Attribute Mapping (20 - 2F) 4-31
4-13. Monochrome Attribute Mapping (30 - 3F) 4-31
4-14. Color Attribute Mapping (00 - 0F) 4-32
4-15. Color Attribute Mapping (10 - 1F) 4-32
4-16. Color Attribute Mapping (20 - 2F) 4-33
4-17. Color Attribute Mapping (30 - 3F) 4-33
4-18. Color Attribute Mapping (40 - 4F) 4-34
4-19. Color Attribute Mapping (50 - 5F) 4-34
4-20. Color Attribute Mapping (60 - 6F) 4-35
4-21. Color Attribute Mapping (70 - 7F) 4-35
4-22. Color Attribute Mapping (80 - 8F) 4-36
4-23. Color Attribute Mapping (90 - 9F) 4-36
4-24. Color Attribute Mapping (A0 - AF) 4-37
4-25. Color Attribute Mapping (B0 - BF) 4-37
4-26. Color Attribute Mapping (C0 - CF) 4-38
4-27. Color Attribute Mapping (D0 - DF) 4-38
4-28. Color Attribute Mapping (E0 - EF) 4-39
4-29. Color Attribute Mapping (F0 - FF) 4-39
4-30. Operations for Updating the Screen Indirectly 4-42
4-31. Operations for Updating the Screen Directly 4-43

8-1. Interprocessor Request Routing Types 8-4

9-1. 8274 and 8530 Register Differences 9-3
9-2. X.21 Status Indication Bit Patterns 9-10

11-1. Mouse Procedures by Function 11-2

2-1. Queue Examples 12-14
12-2. Queue File Header 12-24
12-3. Queue Entry Header 12-27
12-4. Queue Status Block 12-29

13-1. Spooler Scheduling Queue Entry 13-8
13-2. Spooler Status Queue Entry 13-11
13-3. Spooler Control Queue Entry 13-14

14-1. Dial Characters 14-17
14-2. Telephone Service Configuration File Format 14-81
14-3. Telephone Service Configuration Block 14-82
14-4. Telephone Status Structure 14-85
14-5. tUnitState Fields of the Telephone Status Structure 14-87
14-6. lineState Fields of the Telephone Status Structure 14-89
14-7. codecState Fields of theTelephone Status Structure 14-92
14-8. Voice File Header 14-93
14-9. rgMessages Fields of theVoice File Header 14-94
14-10. Voice File Record 14-96
14-11. Voice Control Structure 14-97
14-12. Data Control Structure 14-100

15-1. Performance Statistics Structure 15-17

17-1. ISO Primary Volume Descriptor 17-33
17-2. High Sierra Primary Volume Descriptor 17-42
17-3. ISO Directory Record Format 17-49
17-4. High Sierra Directory Record Format 17-54

Examples

2-1. Special Example 2-9

3-1. Another Example in the Next Chapter 3-4
3-1. ThreePalette.c 3-32
3-2. NewPalette.c 3-40

5-1. Creating a Partition 5-2
5-2. Initializing a New Partition's Environment 5-4
5-3. Loading a Program for Execution 5-6
5-4. Calling a Child-Termination Procedure 5-7
5-5. Child-Termination Procedure 5-8
5-6. Partition Removal Procedure 5-9
5-7. Simple Partition-Managing Program 5-10
5-8. Termination Request Definiton File 5-18

7-1. PLog's Record-Processing Algorithm 7-10

8-1. Sample Request.txt File Using Device Routing 8-7
8-2. Client Program that Supports Device Routing 8-10
8-3. Changes to Illustrate Device Routing 8-12
8-4. Routed Deinstallation Request 8-14
8-5. Sample Request.txt File Using Remote Routing 8-16

10-1. Queue.c 10-35
10-2. Queue.def 10-46
10-3. Dll.mak 10-47
10-4. Queue.h 10-49
10-5. ReadQueue.c 10-50
10-6. WriteQueue.c 10-51
10-7. Clients.mak 10-55
10-8. LinkerConfig.sys 10-57

11-1. Setup 11-6
11-2. Queries 11-10
11-3. Cursor Shape 11-12
11-4. Changing the Graphics Cursor 11-14
11-5. Cursor Movement 11-15
11-6. Sketcher.c 11-17

14-1. Call Progress-Tone 14-21
14-2. Dial.c 14-30
14-3. Response.c 14-36
14-4. Memory.c 14-57
14-5. DataCall.c 14-69

15-1. StatExample.c 15-10

16-1. AsyncService.c 16-38
16-2. Example.c 16-48
16-3. Start.c 16-63
16-4. Stop.c 16-64
16-5. Deinstall.c 16-65

17-1. CdDirectoryList Example 17-11
17-2. CD Search Example 17-14
17-3. Copying a CD-ROM File to Disk 17-19
17-4. Audio Example 17-25

18-1. Fixed Length, Blocked Records 18-21
18-2. Variable Length Records 18-25

About This Guide

Introduction

This two-volume guide provides programming details and coding examples the programmer needs to make full use of the CTOS programming interfaces. It should be used with related CTOS programming documentation.

What This Guide Covers

Volume I of the *CTOS Programming Guide*, "General Programming Topics," explains how to use the programming interfaces to many internal features of the operating system. It focuses primarily on features that may not be common to all versions of the operating system, or to all hardware platforms. For example, the section on multiprocessor system services programming clearly applies only to multiprocessor hardware.

Volume II, "Extended System Services and Libraries," explains the programming interfaces to features that are not internal to the operating system but are packaged with it. For example, the Mouse System Service is a separate system service that a user can choose not to install. However, all users receive the Mouse System Service with their copy of Standard Software.

A separate book, *CTOS/Open Programming Practices and Standards*, should be used with the *CTOS Programming Guide*. *CTOS/Open Programming Practices and Standards* describes the interfaces to features that are common to all CTOS-based operating systems and is a good introduction to CTOS programming. It contains example programs that illustrate how to use various aspects of the CTOS operating system. If your programs follow the guidelines in that book, they should compatible with most versions of CTOS and other CTOS-based operating systems.

Who Should Use This Guide

The *CTOS Programming Guide* is intended for programmers who want to write programs in the CTOS environment. Before using this guide, you should be familiar with the CTOS file system and Executive commands, and you should have some experience programming under CTOS or another operating system (preferably a multitasking one). The *CTOS Programming Guide* should be used in conjunction with *CTOS/Open Programming Practices and Standards.*

How This Guide Is Organized

Volume I of the *CTOS Programming Guide*, "General Programming Topics," is organized as follows:

Section 1. The CTOS Progamming Environment

This section provides a general overview of the CTOS environment as it relates to programmers.

Section 2. Introduction to Protected Mode

This section discusses concepts and features of protected mode processing, the mode most commonly used for CTOS programs. It also provides information for writing protected-mode programs.

Section 3. Using Color

This section describes the use of the ProgramColorMapper operation in defining color for use in graphics and alphanumeric displays.

Section 4. Converting CTOS VGA to Standard VGA

This section describes how to make applications that use CTOS specific VGA attributes work with Standard VGA.

Section 5. Writing Partition-Managing Programs

This section discusses the CTOS methods of creating, using, and deallocating partitions.

Section 6. Software Installation: The Installation Manager

This section discusses how to create installation files that are used by the Installation Manager to install applications and system services.

Section 7. Using the System Log File

This section describes the internal format of the system log file and how applications can use it.

Section 8. Writing Multiprocessor System Services

This section describes how to write system services to run on multiprocessor systems.

Section 9. The Synchronous CommLine Interface

This section describes extensions to the CommLine interface for synchronous data communications.

Section 10. Creating a Dynamic Link Library

This section describes how to create a dynamic link library (DLL).

Volume II of the *CTOS Programming Guide*, "Extended System Services and Libraries," is organized as follows:

Section 11. Mouse Services

This section describes the commands and programs used to access the mouse.

Section 12. Queue Manger

This section discusses the installation, operations, configuration, queue entries, and data structures of the Queue Manager.

Section 13. Spooler

This section describes the spooler configuration, data structures, and operations, and provides programmer's notes on various features.

Section 14. Voice/Data Services

This section describes the telephone services on X-Bus workstations.

Section 15. Performance Statistics Services

This section discusses how an application can use performance statistics services to collect I/O or processing measurements.

Section 16. Asynchronous System Service Model

This section overviews the asynchronous system service model and tells how to use the library procedures contained in *Async.lib*.

Section 17. CD-ROM Service

This section describes the operations used to access CD-ROM media.

Section 18. Sequential Access Service

This section discusses the features of sequential media characteristics and describes sequential access programming techniques.

What Is New in This Guide

The section numbering has changed with this edition. Volume I now contains sections 1 through 10, and Volume II contains sections 11 through 18. In previous editions, section numbering started over in Volume II.

Most of the changes to this edition of the *CTOS Programming Guide* occur in Volume I. New and updated sections include:

Section 4. Converting CTOS VGA to Standard VGA

This is a new section that explains how to convert CTOS specific VGA attributes to Standard VGA attributes.

Section 8. Writing Multiprocessor System Services

This section was previously called "Writing System Services for the XE-530." This section has been updated to include information about EISA multiprocessor systems.

Section 10. Creating a Dynamic Link Library

This is a new section that explains how to write DLL programs.

Note that two sections included in the previous edition of Volume I have been deleted. The section entitled "The SCSI Manager Target Mode" has been deleted because SCSI target mode works only on older hardware. The section entitled "Making CTOS Requests from MS-DOS" has also been removed.

In Section 14, the Audio Service information for SuperGen Series 5000 systems has been removed.

With the exception of the mouse and voice/data examples, the code listings in this guide and the sample programs previously included on an attached disk have been updated to compile using the CTOS MicroSoft C compiler. The diskettes that contain the example programs are now included with the Development Utilities.

Where to Find More Information

The following manuals are part of the CTOS documentation set and may be useful to you. Additional programming tools and libraries are also available. Contact your sales representative for more information.

CTOS Documentation Directory

This reference describes each of the manuals in the set and points the user to specific types of information.

CTOS Executive User's Guide

This manual describes how to use the Executive command interpreter and provides detailed information about command forms and parameter fields for the Standard Software utilities.

CTOS Executive Reference

This reference manual is organized alphabetically by command name. It includes detailed information about the Standard Software commands and special features of the Executive.

CTOS Editor User's Guide

This guide covers the basic use of the Editor to create and modify ASCII text files, and describes new features added to the Editor to enhance its use as a programmer's tool.

CTOS System Administration Guide

This guide contains general information about hardware types and system software products. It provides detailed information about installing system services, user configuration files, formatting disks, backing up data, optimizing performance, configuring and customizing operating systems, and troubleshooting common problems. The bootstrap error list is also included.

CTOS Debugger User's Guide

This guide describes how to use the Debugger commands to debug programs on real-mode and protected-mode operating systems. The manual provides hands-on exercises in using the commands and debugging tips.

CTOS Operating System Concepts Manual

This manual introduces the CTOS III 1.1 operating system and describes the CTOS I 3.4, CTOS II 3.4, and CTOS/XE 3.4 operating systems. It provides an explanation of how the operating system works and gives an orientation to the basic concepts the CTOS programmer needs to understand.

CTOS/Open Programming Practices and Standards

This how-to guide describes the commonly used, hardware independent aspects of programming under CTOS. It covers basic I/O, error handling, parameter management, guidelines for protected-mode programming, writing of nationalizable programs, writing of system services, stack format and calling conventions, mixed-language programming, writing of multiprocess programs, overlays, customized SAM, and communications programming. It includes programming examples.

CTOS Procedural Interface Reference Manual

This alphabetically organized reference manual covers programming operations available with CTOS III 1.1, CTOS II 3.4, CTOS I 3.4, and CTOS/XE 3.4 operating systems, and introduces programming operations new with the CTOS III 1.1 operating system. It also includes descriptions of certain operating system data structures.

CTOS Programming Utilities Reference: Building Applications

This manual explains how to use the Linker, Librarian, Module Definition utility, and Resource Librarian to build applications to run on the CTOS operating system.

CTOS Programming Utilities Reference: Customization

This manual describes how to use the keyboard customization features provided with the CTOS programming utilities. This manual is particularly useful for localizing keyboard support or supporting special keyboard mapping.

CTOS Programming Utilities Reference: Installation and Command Overview

This manual provides installation instructions for the Development Utilities as well as an introduction to the programming utility commands distributed with the CTOS Development Utilities software package. It also provides reference information on each command. This manual is all new for this release.

CTOS Programming Utilities Reference: Assembler

This manual describes how to use the CTOS Assembler. It is meant to be used in conjunction with the Intel CPU Programming References. This manual has not been changed for this release.

CTOS Batch Manager II Installation, Configuration, and Programming Guide

This guide describes the use of the foreground and background Batch products.

CTOS Sort/Merge Programming Reference Manual

This guide describes Sort/Merge utilities and the Sort/Merge object module library.

Section 1
The CTOS Programming Environment

Introduction to CTOS Programming

This section describes the CTOS programming environment and identifies some specific issues that may apply to your environment, depending on which program development tools you use.

This section assumes that you are familiar with CTOS and the Executive. If you are not, see the manuals listed in "About This Guide." If you are new to CTOS, pay particular attention to the *CTOS Executive User's Guide* and the *CTOS Operating System Concepts Manual.*

In addition, the *CTOS/Open Programming Practices and Standards* guide contains extensive programming documentation that covers many of the most popular CTOS operating system features. It is designed to serve as a CTOS primer for programmers and includes program examples.

Distributed Client-Server Computing: The CTOS Cluster

The CTOS cluster is a distributed computing environment based on the client-server model. A cluster consists of one server and one or more client workstations.

When an application program executing on a client workstation calls the operating system, it sends a request to the operating system at that workstation. If the local operating system is unable to provide the requested resource, it automatically forwards the request to the operating system at the server. The server then attempts to provide the requested resource. This process occurs transparently to the application program.

This transparent routing of operating system calls is the foundation of the distributed CTOS environment. It allows applications to take advantage of the CTOS cluster network without any special effort on the part of the application programmer. In effect, any application that runs on a CTOS workstation is a networked application.

For more detailed information about the request mechanism and about related CTOS internals, see the *CTOS Operating System Concepts Manual.*

Tools Available to the CTOS Programmer

An extensive array of program development tools is available under CTOS. The list below describes some of the most commonly used packages.

CTOS Development Utilities

The Development Utilities package is a prerequisite for programming under CTOS. Included in this package are the Assembler, Linker, Librarian, Module Definition Utility, Resource Librarian, and Debugger. It also includes *Ctos.lib*, *CtosToolKit.lib*, and *ResourceLibrarian.lib*, and three libraries of object modules that provide the programming interface to the operating system. Unless otherwise noted, all of the object module procedures documented in the *CTOS Procedural Interface Reference Manual* are included in one of these three libraries.

Language Compilers and Interpreters

In addition to the utility packages, compilers and, in some cases, interpreters are available for the most popular computer languages, including Basic, C, COBOL, FORTRAN, and Pascal.

Additional Programmer's Libraries

While CTOS Development Utilities contain the libraries required for most programming projects, additional libraries are available from Unisys and third-party developers. These libraries provide support for features such as device-independent printing, display of graphics, and interfaces to electronic mail.

Take the time to familiarize yourself with the tools at your disposal, and with the other tools that may be available, before you plunge into programming under CTOS.

An Overview of CTOS System Software

CTOS programs have five methods of invoking system services or library routines: kernel primitives, requests, system-common procedures, dynamic link libraries (DLLs), and object module procedures. The *CTOS Procedural Interface Reference Manual* describes each available operation and specifies its type.

A kernel primitive is a direct command to the CTOS kernel. Kernel primitives can be used when a lower-level interface to the operating system is needed. Most programs use the kernel primitives only indirectly by using the procedural interface for requests.

A request passes through the CTOS kernel, which determines where to route the request for processing. Most internal operating system features are implemented using requests. There is also a special class of application program, called a system service, which serves application-defined requests.

A system-common procedure is a subroutine which is part of the operating system or installed system service and can be called directly. Calls to system-common procedures do not pass through the CTOS kernel. Instead, CTOS uses a feature of the Intel microprocessors (a Call Gate) to redirect the call to the appropriate subroutine within the operating system. As with requests, system-service programs can install their own system-common procedures for use by other applications.

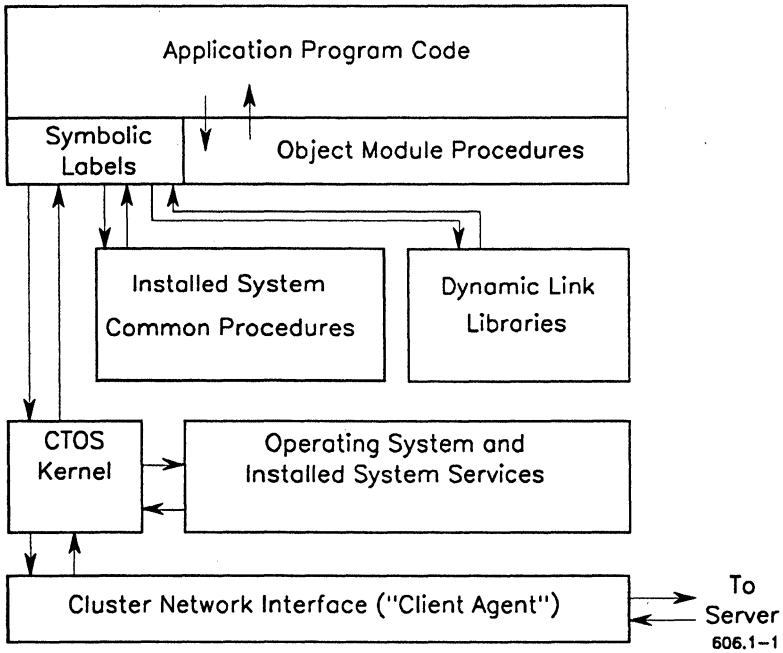
Dynamic link libraries contain procedures that can be shared among several other applications. Unlike conventionally linked libraries whose procedures reside in the caller's run file, DLLs are separate files that are linked to the requesting application program when that program is loaded. Linking modifies the client references to the DLL to point to the DLL as it exists in memory. The DLL contains a list of exports (usually procedures) registered by name. When the client refers to the export address name, the operating system quickly maps the name to the export address. See the "Dynamic Link Libraries" section in the *CTOS Operating System Concepts Manual* for details on how DLLs work. The section entitled "Creating a Dynamic Link Library" in this guide shows you how to write your own DLLs.

An object module procedure is simply a subroutine stored in a library such as *Ctos.lib* or *CtosToolKit.lib*. If your program calls one of those procedures, the Linker links the object module that contains the procedure to your program at link time. That procedure then becomes part of your program.

Figure 1-1 shows the level at which the different types of procedures execute, relative to the CTOS kernel. Notice that the kernel controls access to the cluster network. Therefore, only those system calls that pass through the kernel can be routed across the cluster.

Starting at the top, observe that object-module procedures actually become part of your program when you link it. Calls to these procedures remain "inside" your program. Their code is linked into your program, and your program's thread of execution executes the code.

Figure 1-1. Where CTOS System Calls Are Processed



Shown below the object module procedures are the installed system-common procedures. These procedures are not part of your program, but they do execute on the local processor. Your program's thread of execution executes the code in a system-common procedure, but the code itself is not linked into your program.

On the third level is the CTOS kernel. All requests made by your program are routed through the kernel, which forwards them to the local operating system, to an installed system service (if the appropriate one is available), or to the network. Otherwise, the kernel sends the request over the cluster network to the server for processing.

General Steps in Writing a CTOS Program

The following topics describe the general process of writing and running a program under CTOS. This section assumes you are using a compiled language, such as C. If you are using an interpreted language, such as Basic, the steps may be somewhat different. See your language interpreter manual for more information.

Writing Source Code

Several high-level languages and the assembly language are supported. You can write programs entirely in these languages if they provide all the functionality you need. Your program can also call CTOS operations directly. See your compiler documentation for more information.

Most CTOS programmers use the Editor to write their source code. You can also use other text processing programs, such as OFIS Document Designer. The Editor, Version 11.0 or higher, provides many of the features of a text processor, such as opening multiple files in different windows. In addition, the Editor includes other features designed specifically for the programmer, such as block checking. For details on the Editor, see the *CTOS Editor User's Guide.*

In most languages, programs are written in modules, or subparts, which are linked together later into a complete program. You create a separate source code file for each module.

The CTOS system naming convention is to attach a period and a suffix to the name of a source file. The suffix indicates the language in which the source code is written, for example:

MyProgram.asm	assembly language
MoreCode.pas	Pascal
StringOps.c	C

Usually, you should give your source code text file the same name you will use later for the compiled, object code version of it. You should also use that same file name if your programming language requires you to explicitly name the module inside the source file. Following this convention makes the Linker's map file much easier to read, and helps you keep track of your source code.

Compiling a CTOS Program

After you have written the code, the next step is to compile or assemble the modules of your program. See your compiler documentation for any special requirements.

Most compilers generate the following two output files:

- A *list file* from the compilation. This file is automatically named. Your source file name suffix (if any) is dropped, and the suffix *.lst* is appended to the root name to create the list file name. For example, the list file for *StringOps.c* is named *StringOps.lst*.

  The compiler writes any compilation errors it detects to this file. You can examine the file by using the **Type** command and specifying the list file name in the command form. You can also print it out or edit it.

- An *object file*, if the compilation was successful. The object file contains the executable code generated by the compiler. The name of this file is the root file name with the suffix *.obj*.

Linking a CTOS Program

An object file (or files) created by the compiler cannot be executed. It must first be linked. The Linker accepts one or more object files and creates an executable run file.

To make your program executable, start the Linker, and specify the names of the object modules you want linked, along with the name to give your *run file*. This name, unlike those from the compiler, is not assigned automatically but does default to *.run* if it is not specified. It is conventional to give this file name the suffix *.run*.

The **Link V6** or **Link V8** command starts the Linker and allows you to create run files that execute in either protected mode or real mode. With the **Link V6** command, if you leave the command form field *[Run file mode]* blank or enter the keyword **Real**, the Linker creates a run file that always executes in real mode regardless of the operating system under which it is executed. If you enter the keyword **Protected**, the Linker creates a run file that executes either in protected mode under protected-mode operating systems such as CTOS II or CTOS III, or in real mode under real-mode operating systems such as CTOS I.

The **Link V8** command always creates protected mode programs that will only execute on virtual mode operating systems.

When you use the Linker commands, you may also need to specify other parameters, such as the names of libraries that you want searched or the size of the stack. For details on using the Linker, see the following sources:

- *CTOS Programming Utilities Reference Manual: Building Applications.*
- "Protected Mode Programming Guidelines" in *CTOS / Open Programming Practices and Standards.*
- The section on linking programs in your programming language manual.

If the Link is successful, the following files are generated:

- The *run file* for your program, which is called *.run* if none was specified. (Dynamic link libraries are a type of run file and by convention are name *Dll*RunFileName.*dll*. See the section entitled "Creating a Dynamic Link Library" for details.)
- A *symbol file*, which is used during debugging. It maps the symbolic names you gave your variables to their locations in your program's image in memory. This lets you look up the value of variables by name in the Debugger, instead of having to look up the value of an address, such as DS:04E2h. The symbol file is named RunFileName.*sym* or *.sym* if the RunFileName is not specified.
- A *map file*, which shows the relative address in the memory image and the length of each segment of the program, as well as any errors encountered during the link. The map file has the default name RunFileName.*map* or *.map* if the RunFileName is not specified.

Running the Program

Use the **Run** command to execute your run file. **Run** is a generic command for executing files that do not have Executive commands of their own. The **Run** command simply presents a numbered list of parameter fields, with no special labels to identify runtime parameters.

To give users a conventional means of executing the run file, create a new Executive command for your program using the **Command File Editor** or the **New Command** command. See the *CTOS Executive Reference Manual* for details on using these commands.

Naming Conventions

CTOS programmers are encouraged to use the CTOS naming conventions for variables, procedures, and files. The naming conventions have been designed to improve the readability of the source code by encouraging variable names that are self-explanatory. A well-constructed variable name indicates the variable's use as well as its data type.

Data Types

The majority of the parameters and variables used throughout the CTOS operating system can be declared using one of eight basic data types. In addition, nearly all of the CTOS operating system operations return a status code of data type ErcType. The basic data types, their byte counts, and their descriptions are shown in Table 1-1.

Memory addresses (pointers, selectors, and offsets) are discussed in "Using CTOS Operations" and "Memory Management" in the *CTOS Operating System Concepts Manual.* Logical file addresses are discussed in "File Management" in the *CTOS Operating System Concepts Manual.*

Table 1-1.Basic Data Types

Data Type	Bytes Required	Description
Byte	1	Character or unsigned integer (0-255)
Flag	1	Conditional value (TRUE = 0FFh and FALSE = 0)
ErcType	2	Used by CTOS operations to return status codes
Offset	2	Segment relative address
Selector	2	Memory segment address
Word	2	Unsigned integer (0-65,535)
Lfa	4	Logical file address
Pointer	4	Memory address of the form *selector:offset*
Quad	4	Unsigned integer (0-4,294,967,295)

Defining the Basic Data Types

The following MicroSoft C™ statements define each of the basic CTOS data types listed previously. Once defined, you can use these basic data types to declare variables and structures. To define data types and declare variables and structures in another language, see the documentation for that language.

```
typedef     unsigned char          Byte
typedef     unsigned short int     ErcType
typedef     unsigned char          Flag
typedef     unsigned long int      Lfa
typedef     unsigned short int     Offset
typedef     void _far*             Pointer
typedef     unsigned long int      Quad
typedef     unsigned short int     Selector
typedef     unsigned short int     Word
```

Variable Names

The three basic building blocks of the CTOS naming convention are prefixes, roots, and suffixes. Variable names consist of one or more of these elements. The combination you select depends on the context in which the variable is used. Well-constructed names give an immediate indication of the variable's use and its data type. This results in source code that is much easier to read and maintain.

The naming convention allows great flexibility in combining prefixes, roots, and suffixes. In addition to combinations, variable names can consist of prefixes only (one or more) or roots only (one or more). Suffixes, by definition, accompany the other elements. Examples of common variable name combinations are included at the end of this section. Each of the variable name elements is discussed below.

Prefixes

The prefix identifies the variable function and its data type. Common prefixes and their byte counts, data types, and definitions are shown in Table 1-2. A variable's prefix can be constructed with two or more prefixes. For example, *pcbOutput* is a pointer to a count of bytes. *ppSegmentRet* is a pointer to a location where a pointer to a segment is returned. While prefixes are often useful, they are not always necessary.

Table 1-2. Common Prefixes

Prefix	Bytes	Data Type	Prefix Description
b	1	Byte	Byte (character or unsigned integer).
c	2	Word	Count (unsigned integer).
cb	2	Word	Count of bytes (in a string of bytes).
f	1	Flag	Flag (TRUE must be 0FFh and FALSE must be 0).
i	2	Word	Index (unsigned integer).
l	varies	varies	Literal (a constant).
n	2	Word	Number (unsigned integer) (same as c).
o	2	Offset	Offset from the segment base address.
p	4	Pointer	Logical memory address (pointer) consisting of a segment address and an offset.
pb	4	pointer	Logical memory address of (pointer to) a string of bytes.
q	4	Quad	Quad (unsigned integer).
rg	varies	varies	Array. Usually used with another prefix; for example, the prefix *rgb* identifies an array of bytes.
mp	varies	varies	Map. A one-to-one correspondence between two variables. Usually used with other variables being mapped; for example, *mpcParcRq* maps the count of paragraphs to the count of requests of that size
s	2	Word	Size in bytes (unsigned integer).
sb	varies	varies	String. An array of bytes where the first byte is the size of the string.
sz	varies	varies	String. An array of bytes with a null terminating byte.
w	2	Word	Word.

Roots

The root generally describes the data content of the variable and is usually the most descriptive part of the variable name. A root can be a unique name (such as *clientA*), a general name (such as *erc*), or a combination of a unique name and a general name (such as *exchClientA*). Several common root names, their byte counts, data types, and descriptions are shown in Table 1-3.

Table 1-3. Common Roots

Root	Bytes	Data Type	Prefix Description
erc	2	ErcType	Status code.
exch	2	Word	Exchange.
fh	2	Word	File handle.
fhb	varies*	structure	File Header Block (FHB).
lfa	4	Lfa	Logical file address.
qeh	4	Quad	Queue entry handle.
rq	varies	varies	Request block. Size varies with the request.
sa	2	Selector	Segment address (high-order two bytes of a logical memory address) An sa may be either an sn or an sr. (See below.)
sg	2	Selector	Global Descriptor Table (GDT) selector.
sl	2	Selector	Local Descriptor Table (LDT) selector.
sn	2	Selector	Selector (high-order two bytes of a protected mode memory address).
sr	2	Word	Paragraph number (high-order two bytes of a real mode logical memory address).
userNum	2	Word	User number.

* See "System Structures" in the *CTOS Procedural Interface Reference Manual* for the size of the FHB system structure.

Suffixes

The suffix identifies the use of the variable. Common suffixes include:

Last	Largest allowable index of an array.
Max	Maximum length of an array or buffer (thus one greater than the largest allowable index value).
Ret	Identifies a variable the value of which is set by the called process or procedure rather than by the caller.

Examples of Variable Names

The following are examples of well constructed variable names.

cbFileSpec	Count of bytes in a file specification
ercRet	Status code to be returned to the caller
pbFileSpec	Memory address of a string of bytes containing a file specification
pDataRet	Memory address of an area to which data is to be returned to the caller
ppDataRet	Memory address of a 4-byte memory area to which the memory address of a data item is returned to the caller
pRq	Memory address of a request block
sData	Size (in bytes) of a data area
sDataMax	Maximum size (in bytes) of a data area
psDataRet	Memory address of a 2-byte memory area to which the size of a data item is returned
ssDataRet	Size of the memory area to which the size of a data item is returned

Procedure Naming

The name of a public procedure (one called by modules other than the one in which it is defined) should describe its function and should be specific enough so that it does not conflict with other procedure names. It is a convention to begin each new word or abbreviation within a procedure name with a capital letter, for example:

GetDateTime
PosFrameCursor
QueryDefaultRespExch

Procedures that are part of a system service usually contain an abbreviation that denotes that system service. In the following examples, operations with the "Cd" prefix provide compact disk services, and operations with the "Enls" prefix pertain to Extended Native Language Support:

CdOpen
CdRead
CdSearchFirst
EnlsDrawBox
EnlsFieldEdit
EnlsInsertChar

File Suffix Conventions

The type of a file is denoted by the suffix on its name, as described earlier in this section. Some suffixes are required and some are optional. For those related to programming, compiling, and linking, see the appropriate language manual for more information. Whether required or not, it is a good idea to use the suffixes because they provide useful information about the use of a file.

Table 1-4 shows the most common suffixes. They are related to programming and to system files. Additional suffixes may exist in file names generated by various application programs.

Table 1-4. Common File Suffixes

Suffix	Use
.asm	assembly language source file
.awk	Awk script file
.bas	BASIC source file
.bin	a binary message file
.c	C language source file
.cbl	a COBOL source file
.config	program configuration file
.cpp	C++ language source file
.edf	external definition file
.fls	a file containing a list of files: usually an at-file
.for	FORTRAN source file
.form	object module created by Forms Editor
.gnt	a COBOL intermediate-format file
.h	C-language header file
.idf	internal definition file
.img	bootable protected mode image
.int	a COBOL intermediate-format file
.jcl	a job-control text file
.lib	a library file created by Librarian
.lst	list file from compiler
.map	map file from Linker

continued

Table 1-4. Common File Suffixes (cont.)

Suffix	Use
.mdf	assembly language macro definition file (may contain other definitions also)
.obj	object module from compiler or assembler
.pas	Pascal source file
.plm	PL/M source file
.run	run file from Linker (not assigned by default)
.sed	Sed script file
.sub	submit file, read by Submit command
.sym	binary symbol file from Linker, defining public labels for use by Debugger
.sys	system file, used by operating system
.tmp	temporary file, created by standard software or application program
.txt	a message text file
.user	user configuration file

Models of Segmentation

The Intel architecture uses a number of models of segmentation. These models are based on the way a program is divided. For example, in the small model, all the program's code and data are lumped together in a single segment.

CTOS uses the medium model, in which each object module has its own separate code segment. However, all the global data for the program, plus its stack, are placed together in a single data segment called DGROUP.

Your Compiler and Models of Segmentation

The language compilers and interpreters available from Unisys and from third-party developers use widely varying models of segmentation. Many also support more than one model.

The model you use becomes significant when your program needs to make an operating system call or a library call, because the model your compiler uses may differ from the one CTOS uses. Compilers and interpreters that cannot use the medium model generally provide an interface mediator for operating system calls. See your compiler documentation for more information about how to make operating system calls from that language.

For more general information about models of segmentation, see "Stack Format and Calling Conventions," in the *CTOS/Open Programming Practices and Standards* guide. Note the models of segmentation are also important when using dynamic link libraries. See Appendix D in *CTOS Programing Utilities Reference: Building Applications* for details.

Procedural Calls Between Programming Languages

Calls to the operating system can be thought of as calls to a different programming language. You may also need to call a routine from a program written in a different programming language, such as a C routine from COBOL.

There are several issues involved when you mix programming languages, but the two primary ones are model of segmentation and stack format.

The language compilers and interpreters available from Unisys and from third-party developers generally provide a simple mechanism that allows you to make operating system calls from that language. See your compiler documentation for more information.

For more general information about procedural calls between programming languages, see "Mixed-Language Programming" in the *CTOS / Open Programming Practices and Standards* guide.

Your Compiler and Calls Between Languages

When you need to call nonoperating system procedures written in a different programming language, the best strategy is to map the stack format and the model of segmentation to the CTOS calling convention. The CTOS calling convention is described in "Stack Format and Calling Conventions" in the *CTOS / Open Programming Practices and Standards* guide.

If the language you need to call can use the CTOS calling convention natively, your job is easy. Use the CTOS calling convention for the procedures you need to call and have the calling language routines pretend that they are operating system procedures.

If the language used for the called procedures cannot use the CTOS calling convention, the task is more difficult. Carefully review the documentation for each compiler, and try to identify a calling convention that both can use.

Some Configuration Tips for Programmers

This section collects some miscellaneous information that can help you set up your development system.

The CTOS Debugger

The CTOS Debugger allows you to identify and correct run-time errors in your program without having to add extra output statements to your code. With the Debugger, you can easily check parameters passed to system procedures, looping logic, and protection violations.

When the the CTOS Debugger is installed, its files reside in *[Sys]<Sys>* (where *SysImage.sys* resides). The operating system loads the Debugger into memory automatically when it detects these files. See the *CTOS Debugger User's Guide* for the Debugger configuration that is used with your operating system.

When you are developing programs that run in protected mode, bugs often manifest themselves as protection violations. To trap these violations when they occur, add the following line to your *Config.sys* file:

```
:EnterDebuggerOnFault:Yes
```

This parameter causes the operating system to enter the Debugger whenever it detects a protection violation. You can then trace the cause of the fault before your program exits.

Using Consistent Library Versions

When you update your development environment, take care to maintain its consistency. If you developed a program using one version of *Ctos.lib*, you should keep that version until you want to update the program. Using incompatible versions of software can cause perplexing errors.

One potential problem is that the contents of *Ctos.lib* change from release to release as features are added, and sometimes incompatibilities do occur. For example, a new operating system call could have the same symbolic name as one of the procedures in your program. If the Linker encounters such a situation, it returns an error message.

Another pitfall is accidentally using a symbolic name that is already used in *Ctos.lib*. For example, a program might define a global variable, *FProtectedMode*. That name also corresponds to a procedure in *Ctos.lib*. Such a program will link successfully but will cause a protection fault when it executes.

A third potential pitfall is that a third-party object module library you use could depend on a particular version of *Ctos.lib*. If this type of problem occurs, your program may link successfully but yield unpredictable results when you try to run it.

Finally, when you update your development environment, make sure you install all parts of your new development package. If you install a new version of *Ctos.lib* without new versions of its associated header ("EDF") files, you are likely to run into trouble.

As a general rule, keep close track of the configuration of your development environment and save a copy of your previous library versions when you install new ones.

Section 2
Introduction to Protected Mode

Introduction

This section is intended to familiarize you with protected-mode concepts, so that you will understand how to write programs that run in protected mode on CTOS protected-mode operating systems, such as CTOS II, BTOS II, CTOS III, and CTOS/XE.

This section introduces you to features that are provided by Intel's protected-mode microprocessors and are used by protected-mode CTOS operating systems. For details on Intel microprocessor architecture, see the following Intel manuals:

- *iAPX 286 Programmer's Reference Manual*
- *80386 Programmer's Reference Manual*
- *Intel486 Programmer's Reference Manual*
- *Intel Pentium Programmer's Reference Manual*

In addition, this section compares real-mode addressing to protected-mode addressing. Understanding addressing differences can help you avoid compatibility problems.

To be sure that your programs are compatible (can be loaded and executed in either real or protected mode), follow the guidelines in "Protected Mode Programming Guidelines" in *CTOS/Open Programming Practices and Standards*.

Advantages of Protected Mode

Protected mode offers advantages over real mode, including the following:

- A program can access a much larger address space (memory). The 1-megabyte limit of real mode does not apply.
- The hardware protection mechanism prevents a program from accidentally overwriting code or writing beyond the end of a segment.

Review of Segmented Addressing

On Intel microprocessors, instructions do not accept physical addresses as operands; they accept only segmented logical addresses. A *logical address* is formed from two 16-bit parts, the *segment address* (SA) and the *relative address* (RA). The RA is often referred to as an *offset* from the SA. The processor combines these two parts to identify a location in memory. When using the Assembler or Debugger, logical addresses are written as follows:

SA:RA

As each instruction executes, the processor forms a *linear address* from each logical address. The processor then uses the newly formed linear address to access physical memory. Note that a program never uses a linear address directly. Programs always reference memory by using logical addresses.

The Intel architecture is referred to as a *segmented addressing model*, because of its two-part logical addresses. Every logical address is composed of a relative offset from some segment address (SA). Note the contrast to a *linear addressing model* such as the Motorola architecture, where instructions accept 32-bit linear addresses rather than SA:RA pairs.

When a logical address contains a *real-mode* segment address (SA), that SA is sometimes referred to as an *SR*, to distinguish it from a *protected-mode* segment address, which is sometimes called an *SN*. The abbreviation SA applies to either mode. Thus, in real mode, all logical addresses are actually SR:RA addresses. In protected mode, they are actually SN:RA addresses. The difference between an SR and an SN is described as follows.

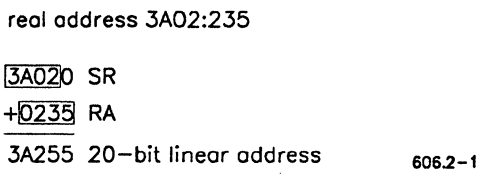
Review of Real-Mode Addressing

Real address mode is the only mode in which 8086 and 80186 microprocessors operate. Protected-mode microprocessors, such as the 80286, 80386, i486, and Pentium, execute in real mode initially at power-up or reset, but then switch to protected mode under control of the operating system. Real-mode addressing is also used when a protected-mode microprocessor executes in virtual 8086 mode.

In real mode, to compute a 20-bit linear address result, the processor shifts the segment address left by four binary places (effectively multiplying it by 16), then adds the relative address.

The resulting 20-bit quantity can address only 1 megabyte of memory (2^{20} locations). For example, the linear address generated from logical address 3A02:235h is shown in Figure 2-1. Note that one hexadecimal digit in the figure corresponds to four binary address bits.

Figure 2-1. Real Mode Segment Addressing



In real mode, each segment address identifies a location 16 bytes higher in memory than the previous segment address. These 16-byte units of memory, each aligned on a 16-byte boundary, are called *paragraphs*. Therefore, a real-mode segment address can be considered a *paragraph number*, because it denotes a particular 16-byte boundary in the physical address space.

In real mode, the *segment registers* contain the paragraph numbers corresponding to the base of the respective segments. The segments are always aligned to start on 16-byte boundaries.

Protected Mode Addressing

Protected-mode microprocessors can address extended memory beyond the first megabyte. The amount of addressable memory is determined by the microprocessor and hardware limits. The 80286 microprocessor, for example, can address up to 16 megabytes of memory in protected mode. To do so requires a linear address of 24 bits. However, no Intel microprocessor can address more than 1 megabyte in real mode. This is because only a 20-bit address can be formed by the address calculation described in the previous section.

To achieve this longer address in a compatible manner and to allow other features of protected mode to be implemented, the same two-part addressing scheme (the segmented addressing model) is used. The address still consists of an SA:RA, but the SA part of the logical address is interpreted differently. The RA part has the same meaning as in real mode.

In protected mode, the 16-bit SA in the segment register is no longer a paragraph number, as it is in real mode. Rather, the SA is an index into a *descriptor table*. The SA is called a *selector* in protected mode because of the indexing or selection function it performs and is written as SN (segment name) to distinguish it from a real-mode segment address.

The operating system constructs and maintains a descriptor table called a *local descriptor table* (LDT) for each run file executing in protected mode from information provided by the Linker in the run file header. Note that to produce the proper run-file format, you must enter the keyword Protected in the **LinkV6** command field, *[Run file mode]*.

Each code and data segment in the program has a unique selector assigned by the Linker and a corresponding unique LDT entry. The LDT is an array of these entries, called *descriptors*, which contain information about the segment.

When an instruction loads a selector into a segment register in a protected-mode program, the processor uses the selector to find the corresponding descriptor and to retrieve a segment base address from it. On the 80286 microprocessor, this base address is 24 bits long; it is 32 bits long on the 80386, i486, and Pentium microprocessors, which have comparatively greater address space (up to 4 gigabytes). When an instruction refers to a memory address (using the segment register and an RA), the RA is added to the base address to obtain a linear address. Unlike real mode, no shift of this base address is done; it is not a paragraph number, but a true physical address.

Figure 2-2 shows this process schematically. Note that the example, 18:235, addresses the same memory location as the example of real mode shown in Figure 2-1.

Figure 2-2. Protected Mode Segment Addressing

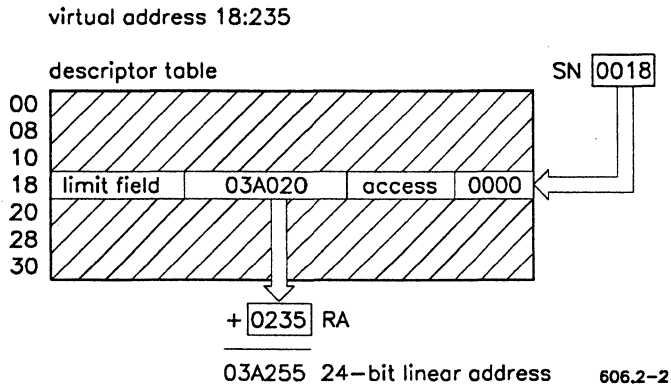


Figure 2-2 shows the descriptor holding the linear base address, 03A020. Instead of the segment register holding the paragraph number 03A02, it holds a selector value, 18, which is used as an index into the descriptor table.

The protected-mode SN:RA address is sometimes referred to as a *virtual address*, because the SN refers only indirectly to memory via a descriptor-table entry. By changing the descriptor table entry, the operating system can make the same virtual address refer to a different linear address for example, to relocate the segment in memory without the program's knowledge.

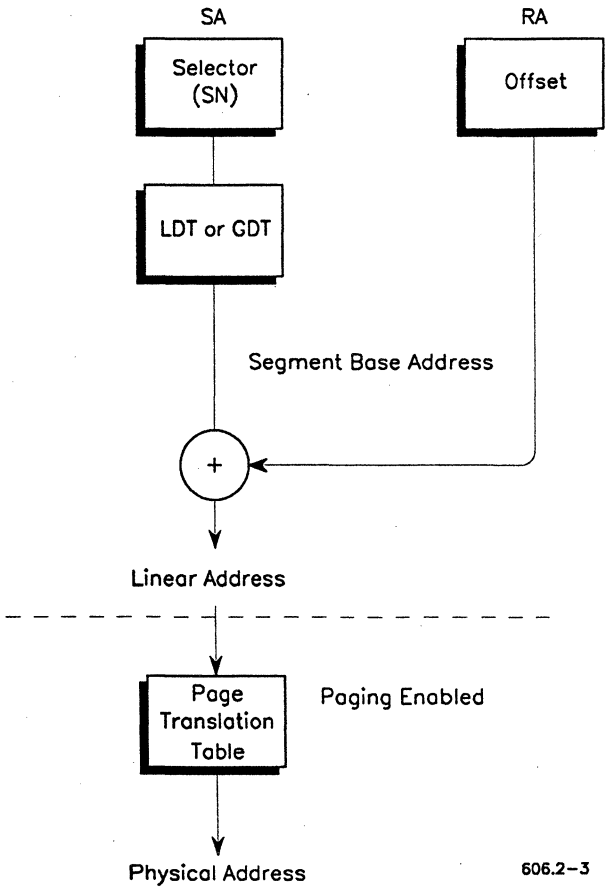
The real-mode SR:RA address is referred to as a *real address*, because it always corresponds to the same linear memory address.

Most incompatibilities between real mode and protected mode arise from this difference between paragraph numbers (SRs) and selectors (SNs). In particular, they arise most often from the assumption that a segment address is a paragraph number.

Memory Mapping Using 80386 Paging

The *physical address* is the actual location in system memory. In real mode and in protected mode without paging, there is a direct correspondence between the linear address and the physical address. When paging is enabled on 80386, i486, or Pentium-based systems in protected mode, however, the linear address is mapped to the physical address by means of a page translation table. The operating system controls the use of this table in a manner transparent to programs. Figure 2-3 illustrates how addresses are mapped.

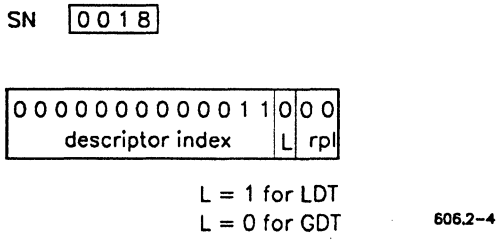
Figure 2-3. Protected Mode Address Translation for Paging



The Format of a Selector

The format of the 16-bit selector (SN) is shown in Figure 2-4. The high-order 13 bits form the offset that indexes into the descriptor table. The next bit discriminates between two kinds of descriptor tables, the local descriptor table (LDT) and the global descriptor table (GDT). The low-order 2 bits (indicated as *rpl* in Figure 2-4) are not currently used by CTOS.

Figure 2-4. The Format of a Selector



The 13 index bits of the selector can generate 2^{13} or 8192 possible values. Thus the hardware permits 8192 entries in each of the two tables.

The Segment Descriptor Tables

Two kinds of descriptor tables are used to address segments: local descriptor tables (LDTs), which were discussed previously, and the *global descriptor table* (GDT). Typically, there is a separate LDT for each run file executing in protected mode. There is only one GDT per processor.

Local Descriptor Table (LDT)

At any one time, there is only one LDT in effect. This LDT is the *current LDT*. It describes the current program's code and data, and the entry points (or call gates) at which the program can make system calls. (See "Gate Descriptors", later in this section.)

Each protected-mode program may be associated permanently with an LDT when it is created. All processes of a program share the same LDT. When process switching occurs, the current LDT is automatically changed by firmware.

Global Descriptor Table (GDT)

The GDT describes global segments (segments that may be addressed by any program or the operating system). The operating system controls the use of the GDT. Currently, CTOS uses GDT-based segments for the following:

- Operating system code and data
- Dynamically allocated system data, such as character maps and the Application System Control Block
- Code and data of programs that contain system-common procedures
- Alias selectors for interprocess communication
- Allocated program memory

Because operating system code and data are GDT-based, the operating system can be called at certain entry points at any time as a subroutine of the user process. The operating system services known as *kernel primitives* and *system-common procedures* are implemented in this manner.

In addition, programs such as the Video Access Method (VAM), which include system-common procedures, must have GDT-based selectors when linked. (Such programs must be linked using the keyword GdtProtected in the **Link V6** or **Link V8** command forms. See the *CTOS Development Utilities Reference Manual: Building Applications* for more information.) A protection mechanism known as a call gate limits the user program to calling at legitimate entry points and provides a convenient way to bind the user program to those entry points at program load time. (See "Gate Descriptors" later in this section.)

When an LDT-based program issues a request to a system service, the operating system gains control and changes the LDT-based pointers in the request block into GDT-based pointers. These GDT-based pointers are called *alias pointers*. *Aliasing* allows the client and the system service to communicate with each other. The system service cannot use the client's LDT-based pointer directly, because the system service also may have an LDT that is different from the client's. Only one LDT can be current at a time.

Figure 2-5 illustrates segment aliasing. Client A issues a request to System Service B. One of the request block parameters is a buffer accessible to Client A at address 84:0. The operating system gains control and creates an alias selector that also points to Client A's buffer. System Service B can access Client A's buffer at the GDT-based address, 100:0.

The Contents of a Segment Descriptor

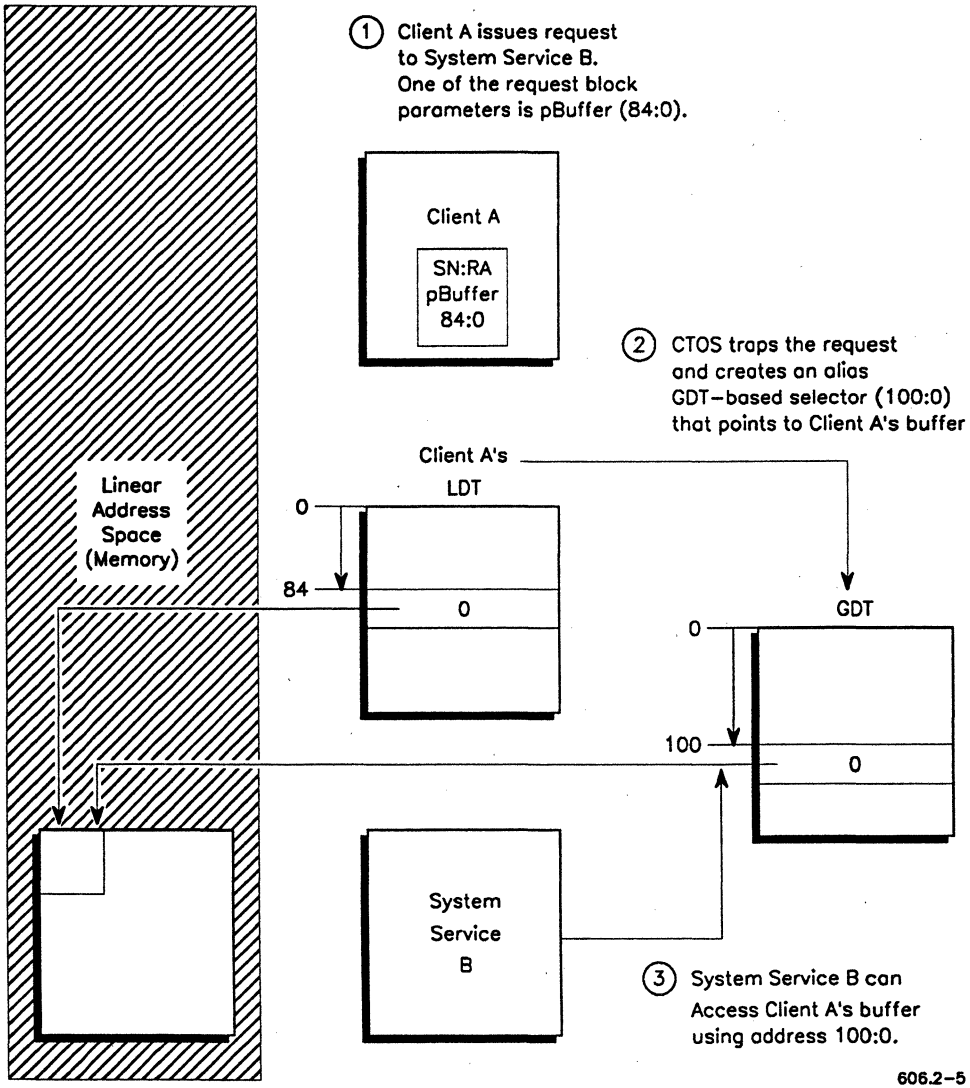
Segment descriptors contain information about individual memory segments. In addition to the base address described in "Protected Mode Addressing," earlier in this section, each segment descriptor contains other information, such as the following:

- The segment's size, which is indicated by its base and limit
- The segment type, such as code or data
- Whether the segment is in memory or is swapped to disk

While a program executes in protected mode, the hardware performs various checks using information in the segment descriptor. These checks provide the following protection features:

- It is not possible to access memory beyond the end of the segment, either accidentally or deliberately.
- Data segments can be designated as writable or readable but can never be used as code segments.
- Code segments are implicitly executable but cannot be overwritten.

Figure 2-5. Pointer Aliasing



Processor Exceptions and Faults

A *fault* occurs if a selector is invalid or is used improperly. Faults transfer control automatically to the operating system.

Faults are of two types: *restartable faults* and *exceptions*. Restartable faults are, in theory, recoverable. Exceptions, on the other hand, are errors that prevent further execution of the program.

The *not-present* fault is an example of a restartable fault. Such a fault occurs when the operating system has marked a segment's descriptor to indicate that it is not currently resident, but is swapped out on disk. This kind of fault gives the operating system the opportunity to read the missing segment into any available free memory, fix up the base address in the descriptor, mark the descriptor *present*, and return to the interrupted program. The program then proceeds to try the instruction that faulted again, since the processor's instruction pointer still points to that instruction.

The use of an invalid selector because of a programming error is an example of an exception, called a *general protection fault*. For instance, some real-mode programs use the ES register to store data. In protected mode, however, only selectors can be loaded into the ES register. If a program loads a value into ES that is not a selector, the hardware's attempt to fetch the associated descriptor results in a fault even before the program attempts to use that value.

There is one exception to this rule. The selector value 0 is special. It can be loaded into a segment register, but it causes a fault if it is subsequently used in an address calculation. The 0 value is permitted to allow the passing of NULL pointers, for example *pbPassword = 0* and *cbPassword = 0* in a CTOS OpenFile call. Prohibiting NULL pointers would be inconvenient, so they are allowed, but the value can never be used in calculating an address.

Additional exceptions can occur later when a valid segment register is used in an address calculation. For example, a *limit exception* occurs when an attempt is made to address beyond the end of a segment.

For more detailed information about faults and exceptions, see the Intel documentation.

Gate Descriptors

A *gate descriptor* is a structure that allows a program to call routines, the addresses of which cannot be known until the program is loaded. Generally, any address that lies outside the run file will be unknown until load time. For example, a program may call a *system-common procedure* or a *kernel primitive*, the location of which the Linker cannot supply.

To resolve such a reference, the Linker uses a *call gate* in the protected-mode run file. At load time, the operating system fills the gate with the virtual address of the called procedure. The actual CALL instructions in the calling code segment, however, are not modified.

In the program code, these calls simply appear to call a virtual address using an ordinary far CALL instruction. When the call is executed, the processor uses the selector to fetch its associated descriptor. Upon examination of the descriptor, however, the processor determines that it identifies a call gate. Therefore, its destination fields are used in place of the original address to reach the appropriate routine. The call gate SN (not the original SN) is placed in the CS register. The original RA is ignored, and a value of zero is placed in the instruction pointer. This address is the first instruction of the called procedure.

In addition to the call gates just described, there are other types of gates with varying functions. *Task state segment* (TSS) gates normally are used by the operating system. TSS gates are also involved in interrupt processing, as are two other gates: *interrupt gates* and *trap gates*. For more information on the function and use of these gates, see the Intel documentation.

Protection Models

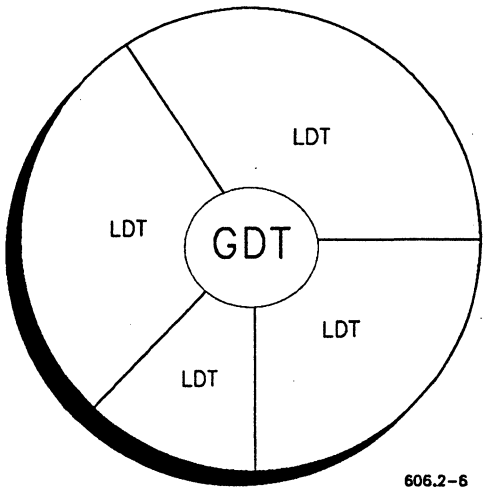
Protected mode on Intel microprocessors actually uses two models of protection. These models are summarized below:

- Protection by *separate address spaces*, shown in Figure 2-6, in which a program is restricted to one virtual address space and cannot describe locations in another address space. CTOS uses this protection model. It provides isolation of one run file from another.

- Protection by *privilege level*, in which every program executes at one of several levels of authority. CTOS III currently uses this protection model to execute real-mode programs in virtual 8086 mode.

In Figure 2-6, each pie slice represents a separate address space, described by a separate LDT (the first protection model). The domain of each descriptor table is also shown.

Figure 2-6. Separate Address Spaces Protection Model



A program in one address space is aware of and can describe only those locations for which it can load selectors. At any given time only one LDT is in effect (the LDT associated with the current process). The GDT is also in effect (all the time). Therefore, each process can address only those objects for which there exist descriptors in its own LDT or the GDT.

The LDT describes the program's own code and data and the gates it may use to make system calls. In other words, it describes only those objects that are legitimate for the program to access. The uses of the GDT are previously described in "Descriptor Tables".

Section 3
Using Color

Introduction

This section describes how to use the ProgramColorMapper operation to define color in graphics and alphanumeric displays. ProgramColorMapper is an object module procedure available in the standard operating system library. Although the single-palette format provided by ProgramColorMapper (described in detail in this section) is available with VAM, version 3.1 and later, the ProgramColorMapper operation itself is backward and forward compatible with all supported versions of the operating system and with Context Manager.

This section begins by discussing a few video concepts you need to understand when manipulating color. The section continues with an overview of ProgramColorMapper and other related color operations that call ProgramColorMapper. Following this introduction, the section describes the structures for setting up and controlling the way color is displayed. At the conclusion of this section, two program examples show how to use ProgramColorMapper to set up palettes in either of two formats.

Before reading this section, familiarize yourself with the video concepts described in "Video" in the *CTOS Operating System Concepts Manual*. In addition, you need to know how to assign attributes to characters and how to display characters to the video device using the VAM and VDM operations. If you are not familiar with these operations, review their detailed descriptions in the *CTOS Procedural Interface Reference Manual*. Finally, if you intend to write graphics applications, you need to be familiar with the graphics library procedures in *Graphics.lib*. See your graphics programming guide for details.

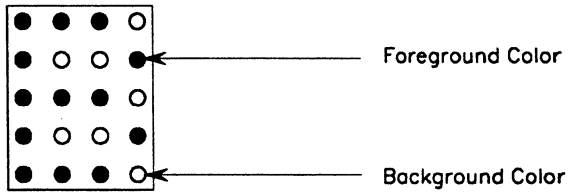
Video Concepts

Brief definitions of video concepts discussed in this section are presented below:

Pixel	An illuminated dot displayed on the video device as a result of manipulating the alphanumeric character map or the graphics multiplane bit map.
Character cell	A rectangular pattern of pixels displayed to the video device. The pixels define a single alphanumeric character and the area surrounding that character. Depending on the hardware, character cells are displayed in different sizes (height and width in pixels) and in different resolutions (number of pixels in the character cell).
Foreground color	Color displayed by the pixels in a character cell that define the character.
Background color	Color displayed by the pixels in a character cell that define the area surrounding the character.

Imagine that the dots shown in Figure 3-1 are two colors: the filled-in dots represent one color, and the empty dots, a different color. All the dots represent pixels defining a character cell. The filled-in dots define the foreground alphanumeric color within the character cell. The empty dots define the background color. Some hardware (for example, workstations with Enhanced Video or VGA) can display multiple background colors. On such hardware, character cells adjacent to each other could display the background in different colors.

Figure 3-1. Character Cell with Foreground and Background Pixels



The concepts of foreground and background need to be taken into account when displaying alphanumeric information through VAM or VDM operations that manipulate the character map (on either character-map workstations or on bit map-workstations with character-map emulation). When calling graphics routines to manipulate the multiplane bit map, however, pixels are not viewed with regard to foreground or background because there is no character-cell concept.

Using the single-palette format through ProgramColorMapper as described in this section, it is possible to use the same palette entry for graphics and alphanumerics. Manipulating color at the level of the pixel requires you to determine which pixel color (foreground, background, or graphics) will ultimately be displayed. (For details, see "Color Priorities," later in this section.)

Program Interfaces

To customize the use of color and to program the graphics control registers, your program must call the ProgramColorMapper operation either directly or indirectly. The operation is implemented in a hardware-independent manner and has run time checks for the support of each workstation type.

All procedures in the standard operating system library that manipulate color call ProgramColorMapper. The SetAlphaColorDefault procedure, for example, calls ProgramColorMapper to establish a default set of colors. SetAlphaColorDefault is called by the Executive as well as by routines in the graphics library. If you do not need to customize the colors you will use, your program can use the default values. Three other color manipulating procedures are documented in the *CTOS Procedural Interface Reference Manual* for backwards compatibility only. These are

SetStyleRam
LoadColorStyleRam
SetStyleRamEntry

Because ProgramColorMapper performs all the functions provided by these procedures, it is recommended that you use ProgramColorMapper in all new programs.

Note that when an application returns control to the Executive, color-palette values that were set within the application are reset to the current Executive default color-palette values. (Other screen attributes, such as font type, resolution, number of lines and columns, reverse video, and character attributes, are also reset to the Executive default values when control returns to the Executive.)

ProgramColorMapper

You use ProgramColorMapper to perform the following functions:

- To enable or disable an alphanumeric or graphics display.
- To display alphanumeric characters in customized colors along with the standard alphanumeric character attributes (half bright, underline, reverse video, blinking, bold, and struck-through). Once you have defined the colors you want to use by calling ProgramColorMapper, you can assign them to the alphanumeric characters you intend to display using VAM or VDM operations, such as PutFrameAttrs, PutFrameChars, and PutFrameCharsAndAttrs.
- To define color for use in graphics displays. If you do not want to use the default color palettes set up by routines in the graphics library, you can use ProgramColorMapper to define your own palettes for use in subsequent calls to graphics library procedures that manipulate the graphics bit map.

Note: *When using standard VGA on EISA/ISA-bus workstations without a CTOS VideoCard, there are certain limitations to normal video capabilities. Color can be substituted for these attributes using CTOS Executive commands. For details, see the "Converting CTOS VGA to Standard VGA" section of this manual. For details on the Executive commands, see the* CTOS Executive Reference Manual.

Once you have defined colors with ProgramColorMapper, you can redefine them by selecting different palettes or different palette entries or by changing the definition of existing entries.

ProgramColorMapper Structures

Through the parameters you provide to ProgramColorMapper, you set up the two types of color structures: a control structure and one or more palettes.

The *control structure* allows you to specify whether you want to enable graphics or alphanumeric mode and whether you want to enable color for either mode. In addition, with the ProgramColorMapper operation available in Standard Software, Version 12.0 and higher, you can specify either of two palette formats. The *single-palette* format sets up a single palette for all color uses; the *three-palette format* allows you use of one to three palettes. For all hardware types, either format can be used, even if the actual color palette hardware is programmed differently.

Each *palette* contains entries that define color for alphanumeric, graphics, and/or background color. Each palette entry defines a color and its intensity. In "Application Notes," later in this section, you will see how color values defined in the palettes combine with the bits in the attribute byte to define the color and attributes of alphanumeric characters.

ProgramColorMapper

- Maintains the control structure and palettes
- Programs the hardware for the palette(s)
- Programs the palette(s) according to the specifications you establish in the control structure

The single-palette format is described first because it provides greater flexibility. Because there is very little difference in the fields of the control structure for either palette format, both control structures are described in "Color Structure," later in this section.

Single-Palette Format

Using the single-palette format, the color palette is a single array of three-word entries.

The words in each entry define color as follows:

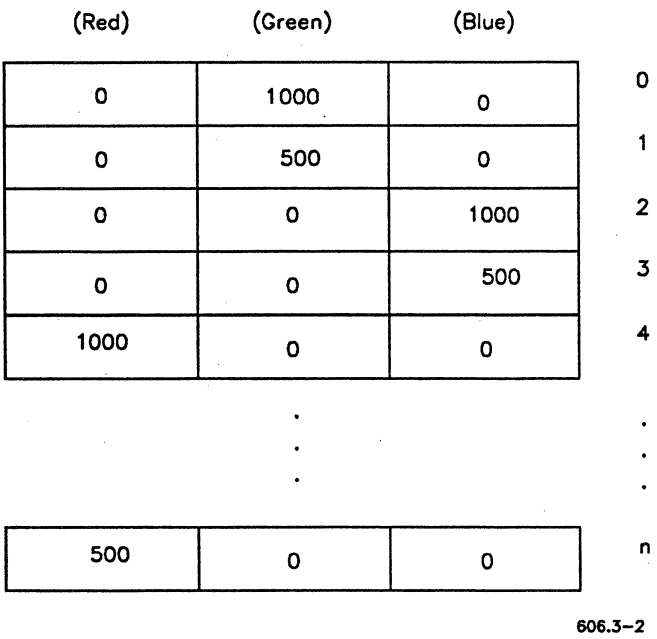
Bits	47-32	31-16	15-0
	Blue	Green	Red

The palette can contain as many entries as the hardware will support (currently up to 256). Through the control structure for this palette format, you can define certain groups of entries for alphanumeric or background color. (See "Color Structure," for details.) Furthermore, any of the entries can be used for graphics color.

To define the intensity of a color, you specify a value in the range of 0 to 1000. Each unit increment in this range represents a one-tenth percent increase in intensity up to one hundred percent intensity. For example, to display a color at fifty percent of its intensity, you can specify a value of 500. A value of 0 means no intensity.

Figure 3-2 shows six entries in an example palette.

Figure 3-2. Example Palette in the Single-Palette Format



In the figure from top to bottom, the palette entries define the following colors:

Full-bright green
Half-bright green
Full-bright blue
Half-bright blue
Full-bright red
Half-bright red

Each palette entry is identified by an index value. (See the numbers to the right of the entries in Figure 3-2.) Index 0 corresponds to the first entry, 1 to the second, and so forth. In a call to ProgramColorMapper, you specify the index value of the first palette entry you want to define in the control structure field *wIndexStart*. (See "Color Structure," later in this section.)

Advantages Over Three-Palette Format

There are several advantages to using the single-palette format over the three-palette format. A few of these are noted here.

Because intensities can be specified in a broad spectrum, this format allows you to be very specific about the intensity of a color. If the hardware supports the intensity, the color intensity is displayed the way you specified. Otherwise the hardware displays the supported intensity closest to the one you chose. If, for example, the hardware supported three levels of intensity, you could specify 330, 670, and 1000 to achieve one-third, two-thirds, and full intensity color, respectively. If you specified 256, the resulting color would be displayed at one-third intensity.

Furthermore, it is relatively easy to specify intensity values: you simply specify percentages. As you will see, this is more user friendly than using the three-palette format to specify intensities. The three-palette format requires that you manipulate bits.

In addition to the wide range of intensities offered by the one-palette format, the control structure allows you to use palette entries interchangeably for different applications.

Three-Palette Format

With this format, color can be defined in three palettes: one palette is an array of alphanumeric character color (*rgbAlpha*), and the other two palettes (*rgbGraphics1* or Graphics Palette 1 and *rgbGraphics2* or Graphics Palette 2) are arrays of graphics colors. Through the control structure for this palette format, you can specify that a graphics palette be used for background color as well. (For details, see "Effect of fBackgroundColor on Palettes" in this section.)

Each palette in this format consists of eight one-byte entries, as shown in the following structure:

Offset	Field	Size (Bytes)
0	rgbAlpha	8
8	rgbGraphics1	8
16	rgbGraphics2	8

Using the three-palette format in a call to ProgramColorMapper, you can assign colors to any of the following:

- A single alphanumeric palette (first eight bytes of the structure)
- A combination of the alphanumeric palette and either Graphics Palette 1 or Graphics Palette 2
- The alphanumeric palette, Graphics Palette 1, and Graphics Palette 2 (all 24 bytes)

The use of these palettes is described below.

Each one-byte palette entry is defined as follows:

Bit	7	6	5	4	3	2	1	0
	A1	A0	R1	R0	G1	G0	B1	B0

Where A is undefined, R is red, G is green, and B is blue.

You specify the intensity of a color by manipulating the bits for that color (such as green, G1 G0) where:

Bit Values	Description
11	Full Intensity
10	2/3 Intensity (normally used for half-bright)
01	1/3 Intensity
00	Off

Color Structure

For each of the palette formats, there is a corresponding control structure. The structure allows programs to set the color in any of the palettes and to turn the alphanumeric character map and the graphics bit map on or off independently.

The control structure for the three-palette format is shown below.

Offset	Field	Size (Bytes)
0	GraphicsPalette	1
1	AlphaEnabled	1
2	AlphaColorEnabled	1
3	GraphicsEnabled	1
4	GraphicsColorEnabled	1

Using the three-palette control structure, programs may switch the graphics bit map to use either of the graphics palettes. This is done by specifying a value for the field *GraphicsPalette*. (See "Field Descriptions," for details.)

The format of a single-palette control structure is an extension of the structure shown above. It contains the following additional fields:

Offset	Field	Size (Bytes)
5	bFormat	1
6	wIndexStart	2

Field Descriptions

The control structure fields are described below (calling ProgramColorMapper with a 0 value in any of the control structure fields causes that field to be left unchanged):

GraphicsPalette — Has no meaning with the single-palette format. With the three-palette format, a value of 1 selects Graphics Palette 1, and a value of 2 selects Graphics Palette 2.

In the next four fields, a value of 1 means enabled, and a value of 2 means disabled.

AlphaEnabled — A value of 1 turns on the alphanumeric character map.

AlphaColorEnabled — With the three-palette format, a value of 1 means the alphanumeric palette can be used for color selection. With the single-palette format, a value of 1 means that palette index entries 8 through 15 can be used for alphanumeric color. (See Figure 3-3.)

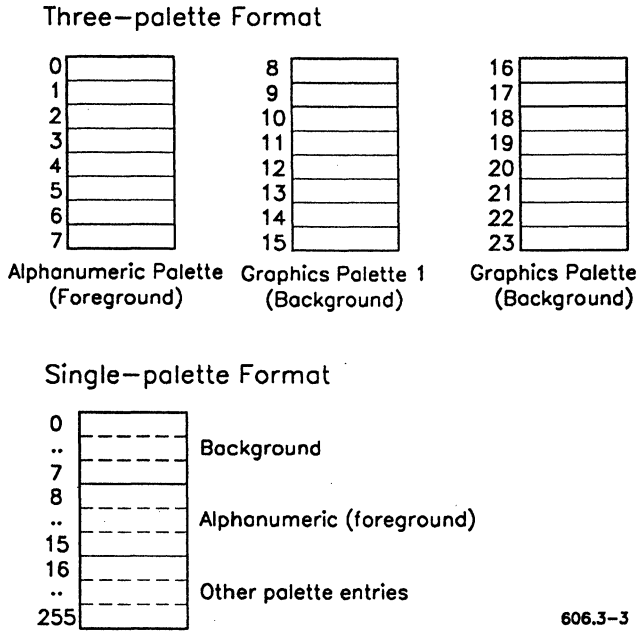
On workstations with background color support, you can set the *fBackgroundColor* flag in a call to the ResetVideoGraphics operation to use background color. (For details on how to define background color in the palettes, see "Effect of fBackgroundColor on Palettes." See "Enabling Background Color Capability," for details on background color configuration.)

A value of 2 displays the default screen colors (for example, green foreground with black background) and enables all normal character attributes.

Note: *When AlphaColorEnabled is 1, the normal character attribute for half bright has its meaning changed. (For details, see "Color Application Notes" in this section.)*

GraphicsEnabled — A value of 1 turns on the graphics bit map.

Figure 3-3. Three-Palette Format and Single-Palette Format



GraphicsColorEnabled — With the three-palette format, a value of 1 means the graphics palette specified by the value of *GraphicsPalette* can be used for color selection. With the single-palette format, A value of 1 means any of the palette entries can be used for graphics color.

bFormat — A value of 1 turns on the single-palette format.

Note: *A value of 0 (or a control structure size of less than 6 bytes) causes ProgramColorMapper to return the three-palette format, even if the palette was previously defined in the single-palette format. In such a case, any value specified for* wIndexStart *(described below) is ignored.*

wIndexStart — Specifies the index value of the first palette entry to be defined or read back using ProgramColorMapper.

Alphanumeric, Background, and Graphics

As previously described, alphanumeric information is displayed through VAM or VDM operations as a rectangular region defining a single alphanumeric character and the area surrounding that character. This region is called a character cell. The font displays in the foreground color and everything else in the cell displays as the background color. Graphics routines simply manipulate pixels without regard to foreground or background because there isn't a character map. The discussion of background color below refers to character-cell background, not what surrounds a graphics object.

Enabling Background Color Capability

Your application can call the ResetVideoGraphics operation to enable and disable background color. Other factors determining whether background color is enabled or disabled are

- Some workstation hardware configurations support background color and some do not. If your workstation configuration does not support background color, the settings described here have no effect.
- If your workstation hardware does support background color (for example, VGA-equipped workstations), you also need to enable background color mode explicitly by adding the following entry to your Config.sys file:

  :EVBackgroundOff: No

The default (Yes) disables background color mode.

Once you have enabled background color mode with the above *Config.sys* entry (and if your hardware supports it), your application can turn the mode on or off by setting or clearing the *fBackgroundColor* flag in a call to the ResetVideoGraphics operation. If selected applications need to use background color, it is recommended that you enable background color mode at the system level (with *Config.sys*). Individual applications can then control whether they want background color by calling ResetVideoGraphics with the appropriate *fBackgroundColor* flag value.

An application can determine the current setting of the *fBackgroundColor* flag by calling the QueryVideo operation.

Effect of fBackgroundColor on Palettes

To be able to use several different background colors on workstations with background color support, your application can set the flag *fBackgroundColor* to TRUE in a call to ResetVideoGraphics. The colors used for background depend on the palette format, as described below.

Background Color and Three-Palette Format

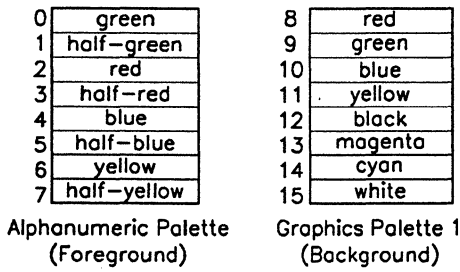
In the three-palette format, you can define background color with either Graphics Palette 1 or Graphics Palette 2. You indicate which of these palettes you want for background color by specifying 1 or 2 for the *GraphicsPalette* field in the control structure. (See Figure 3-3.)

Specifying TRUE for *fBackgroundColor* allows you to use the graphics palette you selected for background color. Each of the eight background colors maps to one of the eight foreground colors in the alphanumeric palette. (See Figure 3-4.) For example, if you selected Graphics Palette 1 for background color (that is, you specified a value of 1 for the *GraphicsPalette* field of the control structure), foreground index entry 0 maps to the first background color (index entry 8) in Graphics Palette 1, foreground index entry 1 maps to the second background color (entry 9), and so forth.

Specifying FALSE for *fBackgroundColor* allows the use of entry 8 only for background color, regardless of the foreground color.

To see how background color affects the three-palette format, let's look more closely at the color assignments shown in Figure 3-4.

Figure 3-4. Sample Three-Palette Format



With *fBackgroundColor* set to TRUE, you obtain the following color combinations for the above palette. The notation used below is *foreground_color / background_color*.

green foreground	/	red background
half-green foreground	/	green background
red foreground	/	blue background
half-red foreground	/	yellow background
blue foreground	/	black background
half-blue foreground	/	magenta background
yellow foreground	/	cyan background
half-yellow foreground	/	white background

With *fBackgroundColor* set to FALSE, using the above palette, you obtain each foreground color (entries 0 through 7) with red (entry 8) as the background color in all cases.

Background Color and Single-Palette Format

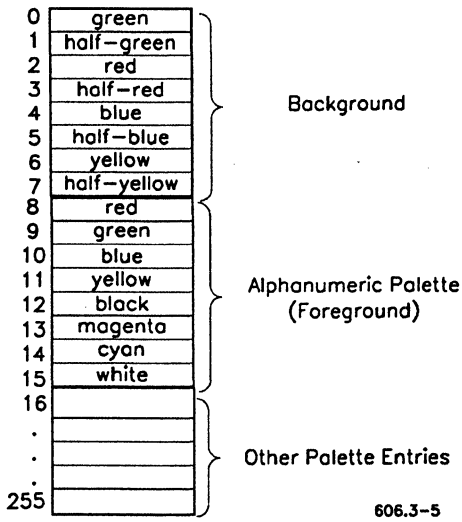
In the single-palette format, you can define background color with palette entries 0 through 7.

Specifying TRUE for *fBackgroundColor* allows you to use all eight of these palette entries for background color. Like the three-palette format, the single-palette foreground colors map to corresponding background colors. (See Figure 3-5.) Foreground index entry 8 maps to background index entry 0, foreground entry 9 maps to background entry 1, and so forth.

Specifying FALSE for *fBackgroundColor* allows the use of entry 0 only for background color, regardless of the foreground color.

To see how background color affects the single-palette format, let's look more closely at the color assignments shown in Figure 3-5.

Figure 3-5. Sample Single-Palette Format



With *fBackgroundColor* set to TRUE, you obtain the following color combinations for the above palette. The notation used below is *foreground_color / background_color*.

red foreground	/	green background
green foreground	/	half-green background
blue foreground	/	red background
yellow foreground	/	half-red background
black foreground	/	blue background
magenta foreground	/	half-blue background
cyan foreground	/	yellow background
white foreground	/	half-yellow background

With *fBackgroundColor* set to FALSE, using the above palette, you obtain each foreground value (entries 8 through 15) with green (entry 0) as the background color in all cases.

More Examples in Three-Palette Format

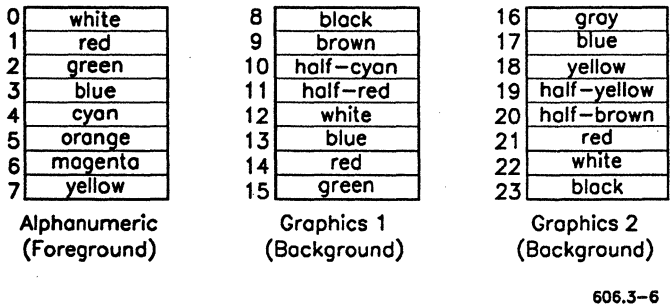
The following examples show the use of the same three-palette format with different settings for the control structure fields.

Case 1: Using Alphanumeric Control Structure Fields

Case 1 manipulates the alphanumeric fields in the control structure. It shows what happens to the palettes when *fBackgroundColor* is set and cleared and the value of the *GraphicsPalette* control structure field is changed.

Figure 3-6 shows palette color assignments for Case 1.

Figure 3-6. Case 1 Palette Settings



Case 1: Condition A

In Condition A, assume that the control structure fields are set as follows (where 1 or 2 for *GraphicsPalette* selects the graphics palette; for all other fields, 1 means enabled, 2 means disabled, 0 means does not change, and x means don't care):

GraphicsPalette	*1*
AlphaEnabled	*1*
AlphaColorEnabled	*1*
GraphicsEnabled	*2*
GraphicsColorEnabled	*x*

Note that with the graphics bit map disabled (*GraphicsEnabled* is 2), it doesn't matter what value is specified for *GraphicsColorEnabled*. The x means the value specified does not have any effect on the palettes.

If *fBackgroundColor* is set to FALSE in this case, the alphanumeric palette is used as the foreground color, and Graphics 1 is used as the background (Graphics 2 is not used). Since background color is off, only entry 0 of Graphics 1 is used for background color. This value is white for white background monitors, and black for black background monitors. In this case, white characters over black background are displayed. The complete set of resulting colors for Condition A is shown below:

white/black
red/black
green/black
blue/black
cyan/black
orange/black
magenta/black
yellow/black

Case 1: Condition B

Now assume *fBackgroundColor* is set to TRUE and the control structure fields are set the same as described for Condition A. In this case, Graphics 1 is used for background colors, as shown below:

white/black
red/brown
green/half cyan
blue/half red
cyan/white
orange/blue
magenta/red
yellow/green

Case 1: Condition C

If *fBackgroundColor* is TRUE and we change the value of the *GraphicsPalette* to 2, Graphics 2 is used for background colors, as shown below:

white/gray
red/blue
green/yellow
blue/half yellow
cyan/half brown
orange/red
magenta/white
yellow/black

Case 1: Condition D

Finally, assume that the control structure fields are set as follows:

GraphicsPalette	*1*
AlphaEnabled	*1*
AlphaColorEnabled	*2*
GraphicsEnabled	*2*
GraphicsColorEnabled	*x*

In Condition D, *AlphaColorEnabled* is disabled. This causes each of the eight foreground colors to be a preset color defined by the system.

If *fBackgroundColor* is FALSE as well, the background color is black (first entry in Graphics Palette 1) and the palette colors are as shown below:

preset system color/black
preset system color/black
preset system color/black
preset system color/black
preset system color/black
preset system color/black
preset system color/black
preset system color/black

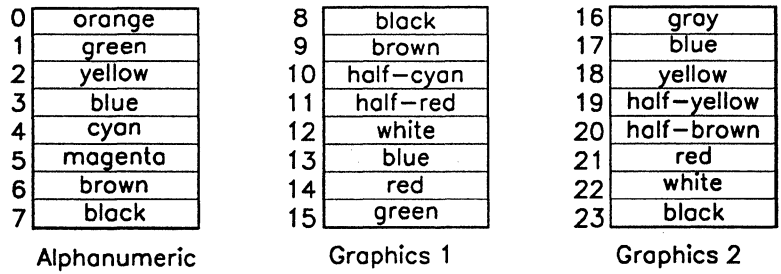
Case 2: Using Graphics Control Structure Fields

Case 2 shows what happens to the palettes as the graphics control structure fields are manipulated and the value of the *GraphicsPalette* control structure field is changed.

Note that in Case 2, the alphanumeric character map is turned off. As a result, the graphics palettes define graphics colors rather than background color in a character cell.

Figure 3-7 shows the Case 2 palette color assignments.

Figure 3-7. Case 2 Palette Settings



Case 2: Condition A

In Condition A, assume that the control structure fields are set as follows (where 1 or 2 for *GraphicsPalette* selects the graphics palette; for all other fields, 1 means enabled, 2 means disabled, 0 means does not change, and x means don't care):

GraphicsPalette	*2*
AlphaEnabled	*2*
AlphaColorEnabled	*x*
GraphicsEnabled	*1*
GraphicsColorEnabled	*2*

Note that with the alphanumeric character map disabled (*AlphaEnabled* is 2), it doesn't matter what value is specified for *AlphaColorEnabled*. The *x* means the value specified does not have any effect on the palettes.

With the above settings, *GraphicsEnabled* turns on the graphics bit map. Graphics color, however, is disabled (*GraphicsColorEnabled* is 2). As a result, your application can perform graphics drawing operations but cannot control the color in which the objects are drawn. The color is preset by the system.

Case 2: Condition B

Next, assume that *GraphicsColorEnabled* is enabled, as shown below:

GraphicsPalette	*2*
AlphaEnabled	*2*
AlphaColorEnabled	*x*
GraphicsEnabled	*1*
GraphicsColorEnabled	*1*

In this case, the resulting drawing colors are specified by Graphics Palette 2.

gray
blue
yellow
half yellow
half brown
red
white
black

Case 3: Using Alpha and Graphics Fields

So far we've discussed how to manipulate the alphanumeric character map and the graphics bit map separately using either the alphanumeric or the graphics fields in the control structure. Oftentimes, however, the hardware allows you to manipulate alphanumeric and graphics color simultaneously. To determine what actually displays requires that there be established color priorities. For details, see "Color Priorities," later in this section.

Maximizing Use of a 16-Entry Palette

If your workstation supports only a 16-entry palette, you can use the single-palette format to define alphanumeric and graphics color by turning off the flag *fBackgroundColor* in a call to ResetVideoGraphics. Then with alpha color enabled, you can use index value 0 for background, index values 1 through 7 for graphics, and index values 8 through 15 for alphanumeric foreground color.

Similarly, with the three-palette format, you could use index values 0 through 7 for alphanumeric color, index value 8 for background, and index values 9 through 15 for graphics color.

Gray-Scale Monitors

Gray-scale monitors interpret the entries in the color palette as shades of gray. Some monitors have very few gray scales. For example, some gray-scale monitors have only black and white; others have only green, half-green, and black. Other gray-scale monitors provide up to 64 shades of gray.

Use the QueryVideo operation to determine the specific capabilities of your monitor and graphics controller, as well as the number of intensities (gray scales) available on your monitor.

Both sample programs at the end of this section can be successfully run on gray-scale monitors that have a reasonable number of gray scales (16 or more) if you comment out the following lines:

```
if (!pVidInfo.fColorMonitor)
   ErrorExitString(0, "This workstation does not support
        color.", 40);
```

Systems with gray-scale monitors work in the same way as do color systems, except that gray shades are calculated based on red, green, and blue values. For systems with fewer than three gray scales, the fields *AlphaColorEnabled* and *GraphicsColorEnabled* in the Color Control structure are always 2 (disabled). For systems with fewer than four gray scales, the field *AlphaColorEnabled* in the Color Control structure is always 2 (disabled).

Color Application Notes

The following paragraphs provide detailed information on how to combine alphanumeric color with the values in the attribute byte, how to define bit-map color, and how to set color priorities.

Combining Alphanumeric Color With the Values in the Attribute Byte

On monochrome workstations or color workstations with color disabled, the attribute assigned to a character with PutFrameAttrs operation is defined as follows:

Bit	Attribute
0	Half-bright
1	Underline
2	Reverse video
3	Blinking
4	Bold
5	Struck-through
6-7	Reserved

Note: *When using standard VGA on EISA/ISA-bus workstations without a CTOS VideoCard, there are certain limitations to normal video capabilities. Manipulating video attributes programmatically, for example, may not result in the display of those attributes or the display may be different than expected. More specifically, blinking has limitations. The underlining, bold, and struck-through attributes are not fully supported. Color can be substituted for these attributes using CTOS Executive commands. For details, see the "Converting CTOS VGA to Standard VGA" section of this manual. For details on the Executive commands, see the* CTOS Executive Reference Manual.

On a workstation with alphanumeric color enabled, the attributes are redefined as follows:

Bit	Attribute
0	p0
1	Underline
2	Reverse video
3	Blinking
4	Bold
5	Struck-through
6	p1
7	p2

p2, p1, and p0 combine to select the index number of one of the palette entries for alphanumeric color (as shown by example in the next few paragraphs). Although the half-bright attribute defined by bit 0 is lost, it is possible to regain half bright by defining half-bright colors.

With either the single-palette or the three-palette format, a maximum of eight entries can define alphanumeric color. Therefore, if you intend to preserve the six character attributes, four alpha colors are possible. Half bright intensity can be defined using four of the entries and full color intensity, using the remaining entries.

Following is an example showing the full-bright and half-bright intensities of four colors using the three-palette format.

Palette Entry	Bit Assignments
0	00001100b (0Ch) (green)
1	00001000b (08h)
2	00110000b (30h) (red)
3	00100000b (20h)
4	00000011b (03h) (blue)
5	00000010b (02h)
6	00111100b (3Ch) (yellow)
7	00101000b (28h)

To make a red character with this palette, you specify an attribute byte to PutFrameAttrs of 01000000b or 40h. The following shows how this is done.

Recall that bits 0, 6, and 7 of the attribute byte combine to select the color index number. Looking at the bits in the attribute byte from left to right, the color index is formed as shown:

Bits in attribute byte	7 6 5 4 3 2 1 0
Bits forming color index	^ ^ ^
Bit values for color index 2	0 1 0

Combining bits 7, 6, and 0, the resulting binary value 010b is the value of the color index. This value selects rgbAlpha(2) of the palette above. The value of rgbAlpha(2) is 00110000b, in which R1R0 is 11 or full bright.

Because no attribute bits are set (bits 1 through 5 for underline, reverse video, blinking, bold, and struck-through, respectively), the value of the attribute byte in binary is

0 1 0 0 0 0 0 0

In hexadecimal, this is 40h.

To make a red, blinking, reverse-video character with the above palette, you specify an attribute byte with bits 2 and 3 turned on as shown below:

Bits in attribute byte	7 6 5 4 3 2 1 0
Bits forming color index	^ ^ ^
	0 1 0 0 0 0 0 0
Bit turned on for reverse video	0 0 0 0 0 1 0 0
Bit turned on for blinking	0 0 0 0 1 0 0 0
Resulting binary value	0 1 0 0 1 1 0 0

In this case the value of the attribute byte is 01001100b or 4Ch.

Note: *With the control structure field* AlphaColorEnabled *set to 1, the single-palette format uses entries in the range of 8 through 15 for alphanumeric color. (See "Field Descriptions," earlier in this section.)*

The call to PutFrameAttrs would be the same whether you defined the eight color entries using the three-palette or the single-palette format. Using the single-palette with *AlphaColorEnabled* set to 1, the combined value of the color bits (2) would select entry 10, the third alphanumeric palette entry.

Defining Bitmap Color

For bitmap color, the graphics planes are combined for each pixel and are used as an index to the correct palette entry for the pixel color.

With the three-palette structure, the resulting value is the index of the entry in the graphics palette (either Graphics Palette 1 or Graphics Palette 2, depending on the value of *GraphicsPalette* in the color control structure). With the single-palette format, *GraphicsPalette* has no meaning. The value directly indexes the entry defining the color.

Note: *ResetVideo leaves the three palettes and the value of* GraphicsPalette *untouched and resets* AlphaEnabled *and* GraphicsEnabled *to 0. SetScreenVidAttrs sets or resets* AlphaEnabled *and leaves the palettes,* GraphicsPalette, *and* GraphicsEnabled *untouched.*

Color Priorities

As indicated earlier in this section, the single-palette format allows you to use any of the palette entries for graphics color. With hardware that allows you to affect pixels using the alphanumeric character map or the graphics bit map, there can be some confusion over which color would actually be displayed. A priority must be set to determine the color displayed. This is the subject of the following example.

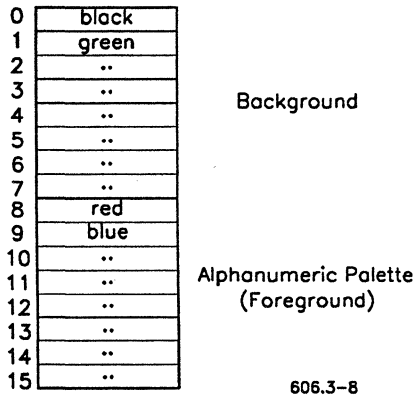
Assume that color is set up in the following way:

- The control structure field *AlphaColorEnabled* is the value 1.
- The flag *fBackgroundColor* was set to TRUE in a previous call to ResetVideoGraphics. Entries 0 through 7 are background colors.
- The values of bits 0, 6, and 7 of the attribute byte combine to select palette entry 1.

P2	P1	P0
0	0	1

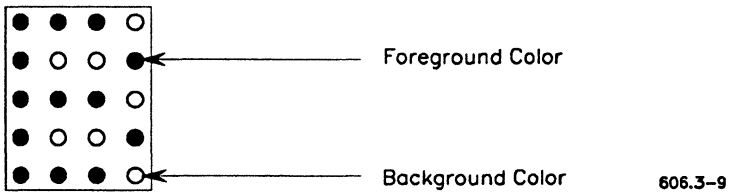
- The palette shown in Figure 3-8 is defined using the single-palette format.

Figure 3-8. Sample Palette



Using this setup, the letter B (shown in Figure 3-9) is displayed to the video device by a call to PutFrameChars. Entry 1 (P2 P1 P0 = 001) indicates blue foreground and green background, since the alphanumeric palette is always foreground and the graphics palette (either Graphics Palette 1 or 2) is background. The letter B is thus displayed with a blue foreground, and background pixels are in green.

Figure 3-9. Color Priorities



Now assume that in addition to the above conditions, *GraphicsEnabled* and *GraphicsColorEnabled* are set to 1 and that you manipulate the pixels through the graphics bit map. If you attempt to display each pixel in the rightmost column shown in the figure using palette entry 8 (red), all the background color pixels in this column would display as red. All the foreground pixels, however, would display as blue.

When combining alphanumerics and graphics to manipulate pixels, the priority (from highest to lowest) of the color displayed is

1. Alphanumeric foreground
2. Graphics
3. Background

Color Programming Tips

The following paragraphs provide useful programming tips.

Avoid Use of Reverse Video and Graphics

Setting the screen to reverse-video mode (with SetScreenVidAttrs) can cause confusing results when used in conjunction with graphics. When alphanumerics are cleared in normal video (not reverse video), nothing exists in the character map, and so it is transparent. This means that any graphics below will be visible.

When alphanumerics are cleared to reverse video, all of the character cells are filled. Since alphanumerics overlay graphics, the graphics will not be visible. It is advisable to use SetScreenVidAttrs to ensure that reverse video is not used in conjunction with graphics.

Querying Single-Palette Format

When you use ProgramColorMapper to return existing palette settings, the three-palette format is returned by default.

To query your existing three-palette format data in single-palette format (and override the default), you need to define a new control structure for the single-palette format. The *bFormat* control byte is not a permanent setting; it must explicitly be set to 1 for single-palette format. Then, in the call to query the palette data, you provide the parameter values for *both* the new control structure and the old palette, as shown below:

```
erc = (ProgramColorMapper(0, 0, pNewControl, 8, pOldPalette,
96, pOldControl, 8));
```

The parameter *pNewControl* points to the control structure in which you set *bFormat* to 1. The size of the structure must be at least 6 bytes to include the *bFormat* field.

Note that the size of *OldPalette* is computed to be 96 bytes. This is because reformatting the entries into the single-palette format results in sixteen 6-byte entries.

Avoid Combining Alpha Background with Graphics

Since palette entries are shared between the alpha background color palette and the graphics palette, the use of alpha background and graphics together is not advised. When background color mode is set and both alpha and graphics are used, graphics palette entries are treated as alpha background colors and may produce unwanted background colors.

Color Program Examples

The following program examples show how to use the ProgramColorMapper operation to display color using the three-palette and single-palette formats. Both sample programs can be successfully run on gray-scale monitors that have a reasonable number of gray scales (16 or more) if you comment out the following lines:

```
if (!pVidInfo.fColorMonitor)
   ErrorExitString(0, "This workstation does not support
                   color.", 40);
```

Three-Palette Format Example

Example 3-1. ThreePalette.c

```
/*
     Program name:  ThreePalette.c

     Description:  This is a sample program that shows how
     three palette style color works for character modes.
     In this example, only the alpha palette is used.
*/

/*
     This ensures that the structure definitions match
     those returned by CTOS.
*/
#pragma pack(1)


/* CTOS type definitions. */

typedef void _far *             Pointer;
typedef unsigned char           Byte;
typedef unsigned short int      Word;
typedef unsigned long int       Quad;
typedef unsigned char           Flag;
typedef unsigned short int      ErcType;

#define ercOK         0
#define FALSE         (0==1)
#define TRUE          (0==0)


#define CheckErc $
#define ErrorExit $
#define ErrorExitString $
#define ProgramColorMapper $
#define PutFrameAttrs $
#define PutFrameChars $
#define QueryVideo $
```

Example 3-1. ThreePalette.c (cont.)

```
#define ReadKbdDirect $
#define ResetFrame $
#define SetAlphaColorDefault $
#include <CTOS.Lib.h>

/*
     Color literals;  NOTE that these are constructed
     using the 'three palette' format:

          A1 A0 R1 R0 G1 G0 B1 B0

     For example, the Magenta color is formed by:

          A1 A1 R1 R0 G1 G0 B1 B0
           0  0  1  1  0  0  1  1 = 33 Hex
*/

#define lBlue            0x03
#define lHalfBlue        0x02
#define lMagenta         0x33
#define lHalfMagenta     0x22
#define lAmber           0x38
#define lOrange          0x34
#define lSalmon          0x25
#define lGrey            0x15

#define lNoAttribute     0x00
#define lReverseVideo    0x04
#define lBlinking        0x08

typedef struct
{
     Byte version;
     Byte nLinesMax;
     Byte nColsNarrow;
     Byte nColsWide;
     Byte graphicsVersion;
     Word nPixelsHigh;
     Word nPixelsWide;
     Word saGraphicsBoard;
```

Example 3-1. ThreePalette.c (cont.)

```
        Word IOPort;
        Word wBytesPerLine;
        Byte nCharHeight;
        Byte nCharWidthNarrow;
        Byte nCharWidthWide;
        Pointer pBitmap;
        Pointer pFont;
        Byte bModuleType;
        Byte bModulePos;
        Word wModuleEar;
        Word nYCenter;
        Word nXCenterNarrow;
        Word nXCenterWide;
        Word wXAspect;
        Word wYAspect;
        Byte bPlanes;
        Byte fColorMonitor;
        Word nColors;
        Byte fBackgroundColor;
        Byte fHardwareCharMap;
        Word nAlternateLinesMax;
        Word nAlternateCharHeight;
        Word wVidRelease;
        Word wVidRevision;
        Byte fGenericColor;
        Byte fVariableCharMap;
        Byte fMultiSystemFonts;
        Byte fUseWsLine;
        Word wLevelsPerPrimary;
        Byte fWhiteBackground;
        Word nExtraRasterLines;
        Byte fStdVGAAttrs;
        Byte Reserved[36];
} VidInfoType;
```

Example 3-1. ThreePalette.c (cont.)

```
typedef struct
{
     Byte      bGraphicsPalette;
     Byte      bAlphaEnabled;
     Byte      bAlphaColorEnabled;
     Byte      bGraphicsEnabled;
     Byte      bGraphicsColorEnabled;
     Byte      bFormat;
     Word      wIndexStart;
} ClrCtrlType;

char *rgbTextDef[] =
{
     "This is green            ",
     "This is half-bright green",
     "This is cyan             ",
     "This is half-bright cyan ",
     "This is white            ",
     "This is half-bright white",
     "This is yellow           ",
     "This is red              "
};
char *rgbTextNew[] =
{
     "This is blue               ",
     "This is half-bright blue   ",
     "This is Magenta            ",
     "This is half-bright Magenta",
     "This is amber              ",
     "This is orange             ",
     "This is salmon             ",
     "This is grey               "
};

/*
     Color attribute byte.  As an example of how to
     interpret the color attribute byte, let's look at
     rgbColorAttrs[6]:
```

Example 3-1. ThreePalette.c (cont.)

```
        If you want to make some text yellow, you need to set
        the color index in the attribute byte to six (6).  The
        bits in the attribute byte that form the color index
        are bits 0, 6, and 7.  Bits 1 through 5 are used for
        the other character attributes (underline, reverse
        video, blinking, bold, and struck-through):

        Attribute byte:                 76543210
             Color index formed          ^^     ^
             from these bits:            ||     |
             To get index 6              11xxxxx0 (bit 0 is the
             1's place, bit 6 is the 2's place and bit 7 is
             the 4's place)
             To get the index 6 mask  11000000 = 0xC0 (hex)

             When using color you lose the half-bright
             attribute.  If you have half-bright colors
             defined, then you can use those as the 'half-
             bright attribute'.
*/

Byte rgbColorAttrs[8] =
{
        0,    1, 0x40, 0x41,
     0x80, 0x81, 0xC0, 0xC1
};


PrintTextWithColor (char **rgbText, Byte bAttrAdded)
{
     Word iCol, iLine;
     Word cbText;
```

Example 3-1. ThreePalette.c (cont.)

```
/*
     Print each line and set the attributes to display the
     appropriate color.  Add in the indicated attribute to
     the color displayed (ie: reverse video).
*/
     CheckErc (ResetFrame (0));
     cbText = strlen(rgbText[0]);
     for (iLine = 0, iCol = 0; iLine < 8; iLine++, iCol++)
     {
          CheckErc (PutFrameChars (0, iCol, iLine,
                    rgbText[iLine], cbText));
          CheckErc (PutFrameAttrs (0, iCol, iLine,
               rgbColorAttrs[iLine] | bAttrAdded, cbText));
     }
}


void main ()
{
     Byte bChar;
     Byte rgbNewPalette[8];
     Byte rgbOldPalette[8];
     VidInfoType pVidInfo;
     ClrCtrlType
          NewControl,
          OldControl;

     CheckErc (QueryVideo (&pVidInfo, 100));

/*
     First check if we can use color.
     NOTE:  This check is not quite complete considering
     that there are analog monochrome monitors that can
     support shades of grey.  Since this is an example of
     color, we won't try to run on monochrome monitors.
*/

     if (!pVidInfo.fColorMonitor)
          ErrorExitString (0, "This workstation does not
               support color.", 40);
```

Example 3-1. ThreePalette.c (cont.)

```
/*
     Save old color palette and control so that we can
     reset upon exit.  We assume that we are being 'run'
     from the Executive.  The Executive uses the 'three
     palette' format so we only need to get 8 bytes for the
     Color Palette and 5 bytes for the Control Structure.
*/

     CheckErc (ProgramColorMapper (rgbNewPalette, 0,
               &NewControl, 0, rgbOldPalette, 8,
               &OldControl, 5));

/*
     Set up default alpha color palette.
*/

     CheckErc (SetAlphaColorDefault (0));

/*
     Display some color text using default palette.
*/
     PrintTextWithColor (rgbTextDef, lNoAttribute);
     CheckErc (ReadKbdDirect (0, &bChar));

/*
     Now use color and reverse video (default palette);
     Note that we are ORing the color attribute byte with
     the Reverse Video attribute (4)
*/
     PrintTextWithColor (rgbTextDef, lReverseVideo);
     CheckErc (ReadKbdDirect (0, &bChar));

/*
     Now let's change the palette
*/

     rgbNewPalette[0] = lBlue;
     rgbNewPalette[1] = lHalfBlue;
     rgbNewPalette[2] = lMagenta;
     rgbNewPalette[3] = lHalfMagenta;
     rgbNewPalette[4] = lAmber;
```

Example 3-1. ThreePalette.c (cont.)

```
        rgbNewPalette[5] = lOrange;
        rgbNewPalette[6] = lSalmon;
        rgbNewPalette[7] = lGrey;
        NewControl.bGraphicsPalette = 2;
        NewControl.bAlphaEnabled = 1;
        NewControl.bAlphaColorEnabled = 1;
        NewControl.bGraphicsEnabled = 2;
        NewControl.bGraphicsColorEnabled = 2;
        CheckErc (ProgramColorMapper (rgbNewPalette, 8,
                  &NewControl, 5, rgbOldPalette, 0,
                  &OldControl, 0));

/*
        Display some color text using new palette.
*/
        PrintTextWithColor (rgbTextNew, lNoAttribute);
        CheckErc (ReadKbdDirect (0, &bChar));

/*
        Now use color and blinking attribute; Note that we are
        ORing the color attribute byte with the blinking
        attribute (8)
*/
        PrintTextWithColor (rgbTextNew, lBlinking);
        CheckErc (ReadKbdDirect (0, &bChar));

/*
        Reset palette and control structure to original
        values.
*/
        CheckErc (ProgramColorMapper (rgbOldPalette, 8,
                  &OldControl, 5, rgbOldPalette, 0,
                  &OldControl, 0));

        ErrorExit (0);
}
```

Single-Palette Format Example

Example 3-2. NewPalette.c

```
/*
     Program name:  NewPalette.c

     Description:  This is a sample program that shows
     how the single palette style color works for
     character modes.

     Caveats:  This is meant to be a sample program
     and should not be considered to be a 'real'
     program.  It is just a representation of basic
     character color concepts.  Also, this program was
     developed for use with the MicrosoftTM C compiler
     for CTOS.  It will most likely require changes
     for other C compilers.
*/

/*
     This ensures that the structure definitions match
     those returned by CTOS.
*/
#pragma pack(1)


/* CTOS type definitions. */

typedef void _far *Pointer;
typedef unsigned charByte;
typedef unsigned short intWord;
typedef unsigned long intQuad;
typedef unsigned charFlag;
typedef unsigned short intErcType;

#define ercOK0
#define FALSE(0==1)
#define TRUE(0==0)
```

Example 3-2. NewPalette.c (cont.)

```
#define CheckErc $
#define ErrorExit $
#define ErrorExitString $
#define InitCharMap $
#define InitVidFrame $
#define ProgramColorMapper $
#define PutFrameAttrs $
#define PutFrameChars $
#define QueryVideo $
#define ReadKbdDirect $
#define ResetFrame $
#define ResetVideoGraphics $
#define SetScreenVidAttr $
#include <CTOS.Lib.h>

#define lNoAttribute   0
#define lReverseVideo  4

typedef struct {
      Byte version;
      Byte nLinesMax;
      Byte nColsNarrow;
      Byte nColsWide;
      Byte graphicsVersion;
      Word nPixelsHigh;
      Word nPixelsWide;
      Word saGraphicsBoard;
      Word IOPort;
      Word wBytesPerLine;
      Byte nCharHeight;
      Byte nCharWidthNarrow;
      Byte nCharWidthWide;
      Pointer pBitmap;
      Pointer pFont;
      Byte bModuleType;
      Byte bModulePos;
      Word wModuleEar;
      Word nYCenter;
      Word nXCenterNarrow;
      Word nXCenterWide;
      Word wXAspect;
```

Example 3-2. NewPalette.c (cont.)

```
        Word wYAspect;
        Byte bPlanes;
        Byte fColorMonitor;
        Word nColors;
        Byte fBackgroundColor;
        Byte fHardwareCharMap;
        Word nAlternateLinesMax;
        Word nAlternateCharHeight;
        Word wVidRelease;
        Word wVidRevision;
        Byte fGenericColor;
        Byte fVariableCharMap;
        Byte fMultiSystemFonts;
        Byte fUseWsLine;
        Word wLevelsPerPrimary;
        Byte fWhiteBackground;
        Word nExtraRasterLines;
        Byte fStdVGAAttrs;
        Byte Reserved[36];
} VidInfoType;

typedef struct
{
        Byte        bGraphicsPalette;
        Byte        bAlphaEnabled;
        Byte        bAlphaColorEnabled;
        Byte        bGraphicsEnabled;
        Byte        bGraphicsColorEnabled;
        Byte        bFormat;
        Word        wIndexStart;
} ClrCtrlType;

typedef struct
{
        Word        wRed;
        Word        wGreen;
        Word        wBlue;
} RGBColorType;
```

Example 3-2. NewPalette.c (cont.)

```
char *rgbText1[] =
{
     "This is green              ",
     "This is half-bright green",
     "This is cyan               ",
     "This is half-bright cyan ",
     "This is white              ",
     "This is half-bright white",
     "This is yellow             ",
     "This is red                "
};
char *rgbText2[] =
{
     "Text green/Background black                          ",
     "Text half-bright green/Background grey               ",
     "Text cyan/Background blue                            ",
     "Text half-bright cyan/Background half-bright blue  ",
     "Text white/Background magenta                        ",
     "Text half-bright white/Background half-bright
          magenta",
     "Text yellow/Background half-bright red               ",
     "Text red/Background half-bright yellow               "
};
char *rgbText3[] =
{
     "Text black/Background green                          ",
     "Text grey/Background half-bright green               ",
     "Text blue/Background cyan                            ",
     "Text half-bright blue/Background half-bright cyan  ",
     "Text magenta/Background white                        ",
     "Text half-bright magenta/Background half-bright
          white",
     "Text half-bright red/Background yellow               ",
     "Text half-bright yellow/Background red               "
};

/*
     Color attribute byte.  As an example of how to
     interpret the color attribute byte, let's look at
     rgbColorAttrs[6]:
```

Example 3-2. NewPalette.c (cont.)

```
      If you want to make some text yellow, you need to set
      the color index in the attribute byte to six (6).  The
      bits in the attribute byte that form the color index
      are bits 0, 6, and 7.  Bits 1 through 5 are used for
      the other character attributes (underline, reverse
      video, blinking, bold, and struck-through):

      Attribute byte:              76543210
           Color index formed       ^^     ^
           from these bits:         ||     |
           To get index 6           11xxxxx0 (bit 0 is the
           1's place, bit 6 is the 2's place and bit 7 is
           the 4's place)
           To get the index 6 mask  11000000 = 0xC0 (hex)

           When using color you lose the half-bright
           attribute.  If you have half-bright colors
           defined, then you can use those as the 'half-
           bright attribute'.
*/

Byte rgbColorAttrs[8] =
{
      0, 1, 0x40, 0x41, 0x80, 0x81, 0xC0, 0xC1
};


/*
      This macro fills in the color palette entries (hues)
      for the RED, GREEN, and BLUE parts of the indicated
      color palette index.
*/
#define SetColorHues(iCOLOR,RED,GREEN,BLUE) \
      rgColorPalette[iCOLOR].wRed = RED; \
      rgColorPalette[iCOLOR].wGreen = GREEN; \
      rgColorPalette[iCOLOR].wBlue = BLUE
```

Example 3-2. NewPalette.c (cont.)

```
PrintTextWithColor (char **rgbText, Byte bAttrAdded)
{
     Word iCol, iLine;
     Word cbText;

/*
     Print each line and set the attributes to display the
     appropriate color.  Add in the indicated attribute to
     the color displayed (ie: reverse video).
*/
     CheckErc (ResetFrame (0));
     cbText = strlen(rgbText[0]);
     for (iLine = 0, iCol = 0; iLine < 8; iLine++, iCol++)
     {
          CheckErc (PutFrameChars (0, iCol, iLine,
                    rgbText[iLine], cbText));
          CheckErc (PutFrameAttrs (0, iCol, iLine,
                    rgbColorAttrs[iLine] | bAttrAdded,
                    cbText));
     }
}


Flag FEnableBackgroundColor (Flag fBackgroundOn, Word nCols,
                             Word nLines)
{
     Word sCharMap;
     VidInfoType VidInfo;

/*
     Enable/Disable background color by resetting the
     screen.  Then query to see if background color was
     actually enabled.
*/
     CheckErc (ResetVideoGraphics (nCols, nLines, TRUE,
               0x20, &sCharMap, 0, 0, 0, fBackgroundOn));
     CheckErc (InitVidFrame (0, 0, 0, nCols, nLines, 0, 0,
               0, FALSE, FALSE));
     CheckErc (InitCharMap (0, sCharMap));
     CheckErc (SetScreenVidAttr (1, TRUE));
```

Example 3-2. NewPalette.c (cont.)

```
    CheckErc (QueryVideo (&VidInfo, sizeof(VidInfo)));
    return (VidInfo.fBackgroundColor);
}


void main ()
{
    Byte rgbOldPalette[8];
    Byte bChar;
    Flag fBackgroundColor;
    Word nCols;
    Word nLines;
    RGBColorType rgColorPalette[16];
    VidInfoType VidInfo;
    ClrCtrlType
        NewControl,
        OldControl;

/*
    Save the current state of the screen.  Use the current
    screen dimensions and resolution later.
*/
    CheckErc (QueryVideo (&VidInfo, sizeof(VidInfo)));
    nCols = VidInfo.nColsNarrow;
    nLines = VidInfo.nLinesMax;

    if ((VidInfo.wVidRelease < 3) ||
        ((VidInfo.wVidRelease == 3) &&
        (VidInfo.wVidRevision < 2)))
        ErrorExitString (0, "This program requires VAM
        version 3.2 or later.", 47);

/*
    First check if we can use color.
    NOTE:  This check is not quite complete considering
    that there are analog monochrome monitors that can
    support shades of grey.  Since this is an example of
    color, we won't try to run on monochrome monitors.
*/
```

Example 3-2. NewPalette.c (cont.)

```
    if (!VidInfo.fColorMonitor)
         ErrorExitString (0, "This workstation does not
              support color.", 40);

/*
    Save old color palette and control so that we can
    reset upon exit.  We assume that we are being 'run'
    from the Executive.  The Executive uses the 'three
    palette' format so we only need to get 8 bytes for the
    Color Palette and 5 bytes for the Control Structure.
*/
    CheckErc (ProgramColorMapper (0, 0, 0, 0,
    rgbOldPalette, 8, &OldControl, 5));

/*
    Start setting up new color palette;  We are making
    this color palette look like default color palette as
    set up by the SetAlphaColorDefault call.  NOTE that
    foreground colors start at index 8.
*/

    /*              RED GREEN BLUE    Background colors  */

    SetColorHues( 0,    0,    0,    0); /* Color 0 - Black */
    SetColorHues( 1, 333, 333, 333); /* Color 1 - Dark
                                        gray */
    SetColorHues( 2,    0,    0,1000); /* Color 2 - Blue */
    SetColorHues( 3,    0,    0, 500); /* Color 3 - Half-
                                        bright blue */
    SetColorHues( 4,1000,    0,1000); /* Color 4-Magenta */
    SetColorHues( 5, 500,    0, 500); /* Color 5 - Half-
                                        bright Magenta */
    SetColorHues( 6, 500,    0,    0); /* Color 6 - Half-
                                        bright red */
    SetColorHues( 7, 500, 500,    0); /* Color 7 - Half-
                                        bright yellow */
```

Example 3-2. NewPalette.c (cont.)

```
        /*              RED GREEN BLUE    Foreground colors  */

        SetColorHues( 8,    0,1000,    0); /* Color 0 - Green */
        SetColorHues( 9,    0, 500,    0); /* Color 1 - Half-
                                              bright green */
        SetColorHues(10,    0,1000,1000); /* Color 2 - Cyan */
        SetColorHues(11,    0, 500, 500); /* Color 3 - Half-
                                              bright Cyan */
        SetColorHues(12,1000,1000,1000); /* Color 4 - White */
        SetColorHues(13, 500, 500, 500); /* Color 5 - Half-
                                              bright white */
        SetColorHues(14,1000,1000,    0); /* Color 6 -Yellow */
        SetColorHues(15,1000,    0,    0); /* Color 7 - Red */

/*
        Enable alpha color only and set up the color palette.
*/
        NewControl.bGraphicsPalette = 2;
        NewControl.bAlphaEnabled = 1;
        NewControl.bAlphaColorEnabled = 1;
        NewControl.bGraphicsEnabled = 2;
        NewControl.bGraphicsColorEnabled = 2;
        NewControl.bFormat = 1;
        NewControl.wIndexStart = 0;

/*
        Set new color palette.  Note that the size of the
        color palette is: 3 Words (Red, Green, Blue) times 2
        bytes per word times the number of entries in the
        palette (16) = 96.
*/
        CheckErc (ProgramColorMapper (rgColorPalette, 96,
                  &NewControl, 8, 0, 0, 0, 0));

/*
        First display the information without background color
        on.
*/
```

Example 3-2. NewPalette.c (cont.)

```
    fBackgroundColor = FEnableBackgroundColor (FALSE,
              nCols, nLines);
    if (!fBackgroundColor)
    {
         /* Display some color text */
         PrintTextWithColor (rgbText1, lNoAttribute);
         CheckErc (ReadKbdDirect (0, &bChar));

         /* Now use color and reverse video. */
         PrintTextWithColor (rgbText1, lReverseVideo);
         CheckErc (ReadKbdDirect (0, &bChar));
    }

/*
    Now display the color information with background
    color on.  If background color is not supported, skip
    this part.
*/
    fBackgroundColor = FEnableBackgroundColor (TRUE,
              nCols, nLines);
    if (fBackgroundColor)
    {
         /* Display some color text with background color
*/
         PrintTextWithColor (rgbText2, lNoAttribute);
         CheckErc (ReadKbdDirect (0, &bChar));

         /* Now use color and reverse video. */
         PrintTextWithColor (rgbText3, lReverseVideo);
         CheckErc (ReadKbdDirect (0, &bChar));
    }

/*
    Reset to original state;
*/
    (void) FEnableBackgroundColor
         (VidInfo.fBackgroundColor, nCols, nLines);
    CheckErc (ProgramColorMapper (rgbOldPalette, 8,
         &OldControl, 5, 0, 0, 0, 0));

    ErrorExit (0);
}
```

Section 4
Converting CTOS VGA to Standard VGA

Overview

With the release of Video Access Method (VAM) 4.1.5, most CTOS character-based applications are supported on workstations that use the Standard Video Graphics Array (VGA) display adapter. Standard VGA is the video hardware used in EISA/ISA-bus workstations. Because CTOS graphics applications are not supported on the Standard VGA hardware, this section addresses character-based applications exclusively.

Prior to VAM 4.1.5, most CTOS applications that ran on EISA/ISA-bus workstations had to use the CTOS Videocard to maintain the same level of CTOS video capability as could be attained by running these applications on workstations using proprietary CTOS VGA hardware. With VAM 4.1.5, most of these existing applications can run effectively on EISA/ISA-bus machines without the CTOS Videocard.

This section contains guidelines on how to convert such applications to work with Standard VGA hardware without the CTOS Videocard. As a general rule, we are encouraging developers to modify their applications to take advantage of the capabilities of Standard VGA. To provide backward compatibility, however, VAM 4.1.5 has been enhanced to support a significant number of existing CTOS applications. This section describes these enhancements and recommends ways to handle CTOS VGA features that do not directly translate to Standard VGA.

This section refers to the standard VGA adapter as Standard VGA to distinguish this hardware from proprietary CTOS VGA hardware. CTOS Standard VGA VAM is packaged with VAM 4.1.5.

This section addresses CTOS VGA and Standard VGA differences in three areas: VGA font capabilities, display resolution, and character attributes.

For each of these areas, this document

- Identifies how CTOS VGA and Standard VGA differ
- Identifies the types of applications affected by this difference
- Recommends ways to modify affected applications
- Explains how the VAM 4.1.5 (CTOS Standard VGA VAM) enhancements are implemented to protect as many existing applications as possible from this video hardware difference

Who Should Use This Information

If you are moving a CTOS character-based application to Standard VGA hardware, ask yourself the following questions. Does your application

- Use customized fonts?
- Require a special display resolution?
- Use the bold or struck-through attribute?
- Access the character map at an interface level below VAM?

If you answer yes to any of these questions, read this document. It tells you how you may need to modify your application.

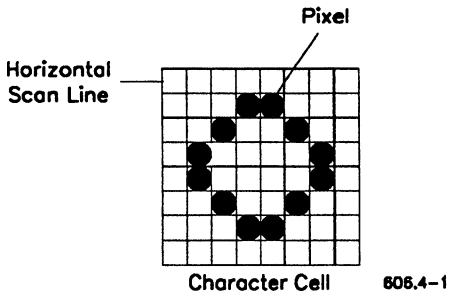
General Concepts and Terminology

We assume that you have written CTOS character-based applications and have some familiarity with video concepts. The paragraphs below briefly describe some standard terms this document uses. To present the terms in context, we provide some conceptual background. If you need additional information, we recommend that you consult documentation on the Standard VGA adapter. If you understand the following concepts, you can skip this discussion completely.

The video hardware represents a character as a rectangular pattern of illuminated dots or *pixels* that it displays to the screen as a *character cell*. (See Figure 4-1.) Each pixel is represented by a bit in memory.

A *font* is a collection of characters and symbols sharing a common height and similar appearance.

Figure 4-1. Character Cell

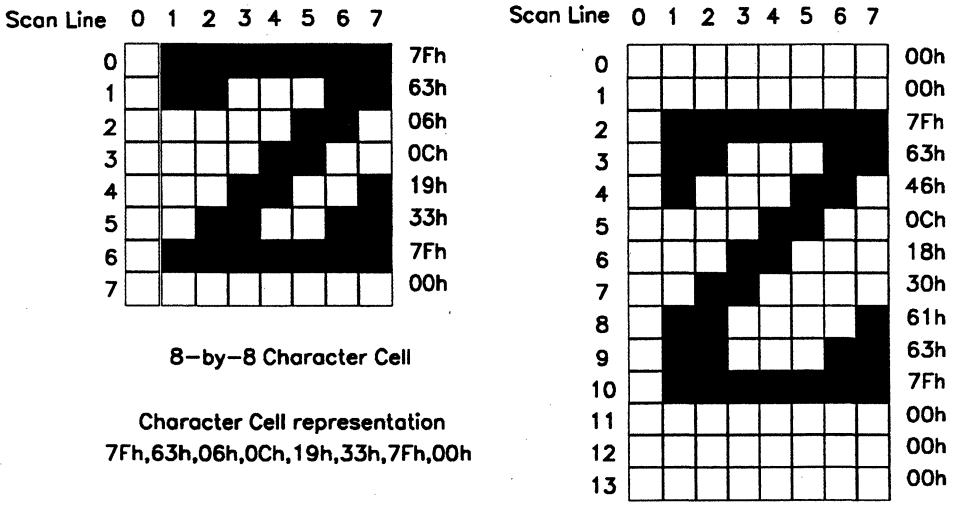


A character is represented by two bytes: a one-byte *character code* (0-255) and a character *attribute byte*. The character code is the ASCII representation of the character. The lowercase letter a, for example, is 61h. (See Appendix B of the *CTOS Procedural Interface Reference Manual* for a list of the character codes for all standard and nationalized characters.) The attribute byte controls the character presentation on the screen. Attributes include reverse video, underlining, and bold.

In a character cell, each *horizontal scan line* is represented as a sequence of bytes. (See Figure 4-2.) Each bit in each byte controls the illumination of a pixel. The first byte corresponds to the top scan line of the character, and the last byte corresponds to the bottom scan line. The most significant bit in each of these bytes is the far left pixel of the character.

Foreground color is displayed by the pixels defining the character. Any bit set to 1 displays the foreground color. *Background* color is displayed by the pixels surrounding the character. Background color pixels are displayed by the bits set to 0.

Figure 4-2. Scan Lines

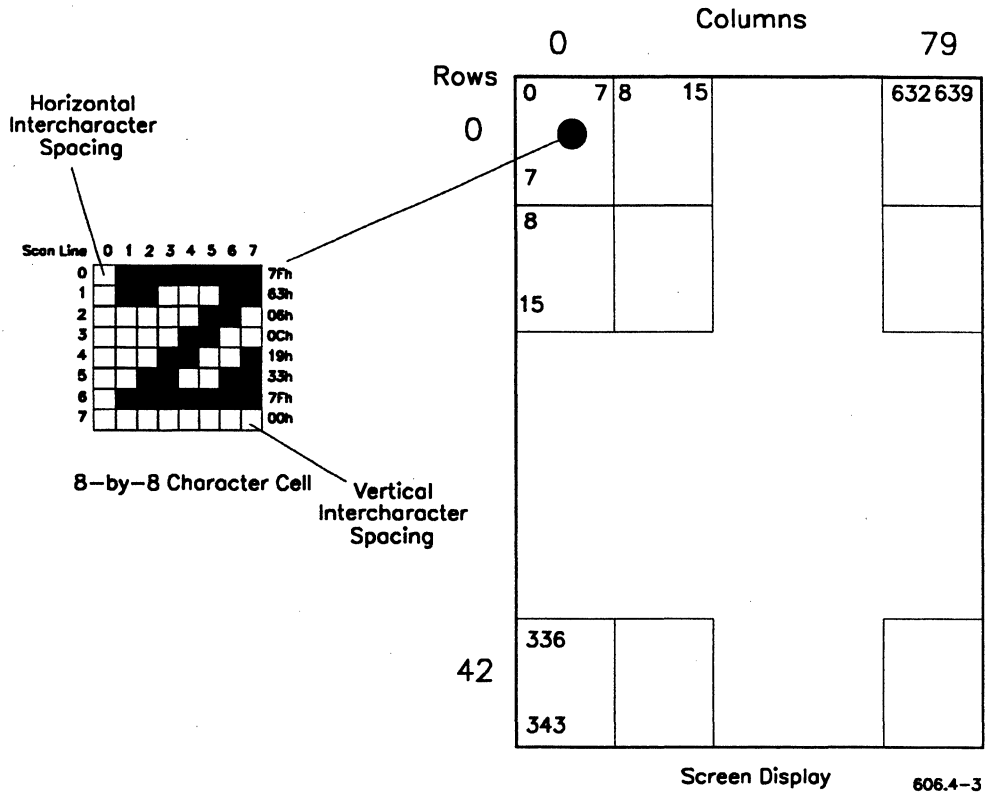


606.4-2

Character cells appear on the screen in a tiled fashion. No space exists between cells, and cells do not overlap. To distinguish one character cell from those bordering it, each character cell must include horizontal and vertical *intercharacter space*. Normally, under CTOS, this intercharacter space occurs at the bottom and to the left of the character.

Figure 4-3 shows the relationship of a character cell to a screen display. The particular screen display shown consists of 43 rows by 80 columns of character cells. The intercharacter spacing separates an individual character from those surrounding it.

Figure 4-3. Relationship of a Character Cell to a Screen Display



VGA Font Capabilities

Font capabilities is an area in which CTOS VGA differs from Standard VGA. CTOS VGA supports two font-related features not available on Standard VGA. These are

- Nine-bit-wide fonts
- Half-bit shift

If your application loads customized fonts, it will be affected by the move to Standard VGA hardware.

As a general rule, we are recommending that developers provide a character cell size of 8-by-16 pixels for their customized fonts. However, the CTOS Standard VGA VAM will continue to load CTOS 9-by-12 pixel font files. VAM automatically converts the 9-by-12 font to 8-by-12. The extent to which this conversion is successful depends upon the particular font.

This section provides a detailed description of the above font-related issues.

Nine-Bit-Wide Fonts

Both CTOS VGA and Standard VGA support nine-bit-wide character cells. However, there are significant differences between the two implementations, as noted below.

Standard VGA Capability

With Standard VGA, only eight bits of the nine-bit-wide character cell are actually *stored* in the font. The hardware determines how to display the ninth bit based on the character's position in the character set. For character codes 0 through 127 (7Fh), the hardware assumes the ninth bit to be a horizontal intercharacter space. For character codes 128 (80h) through 255 (FFh), the hardware copies the eighth bit into the ninth bit position.

Figure 4-4 illustrates how the Standard VGA hardware represents the character 'm' (6Dh) (a character between 0 and 127) on the screen.

Figure 4-4. Characters Codes 0 through 127

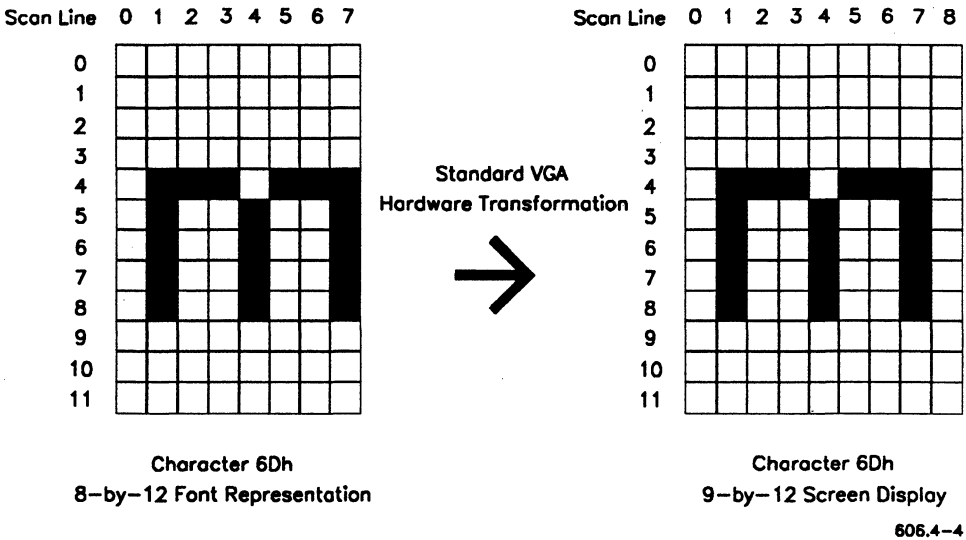
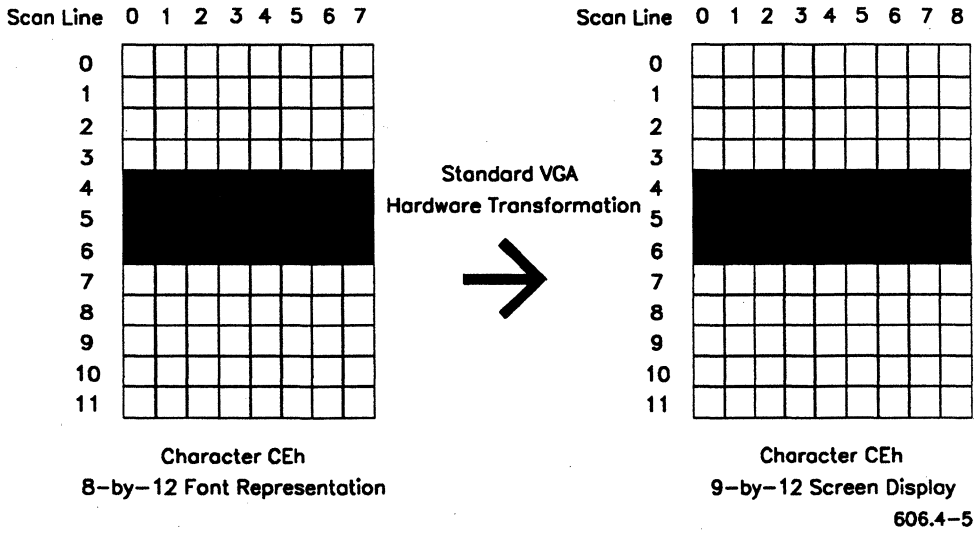


Figure 4-5 illustrates how the Standard VGA hardware represents the character CEh (a character between 128 and 255).

Figure 4-5. Characters Codes 128 through 255

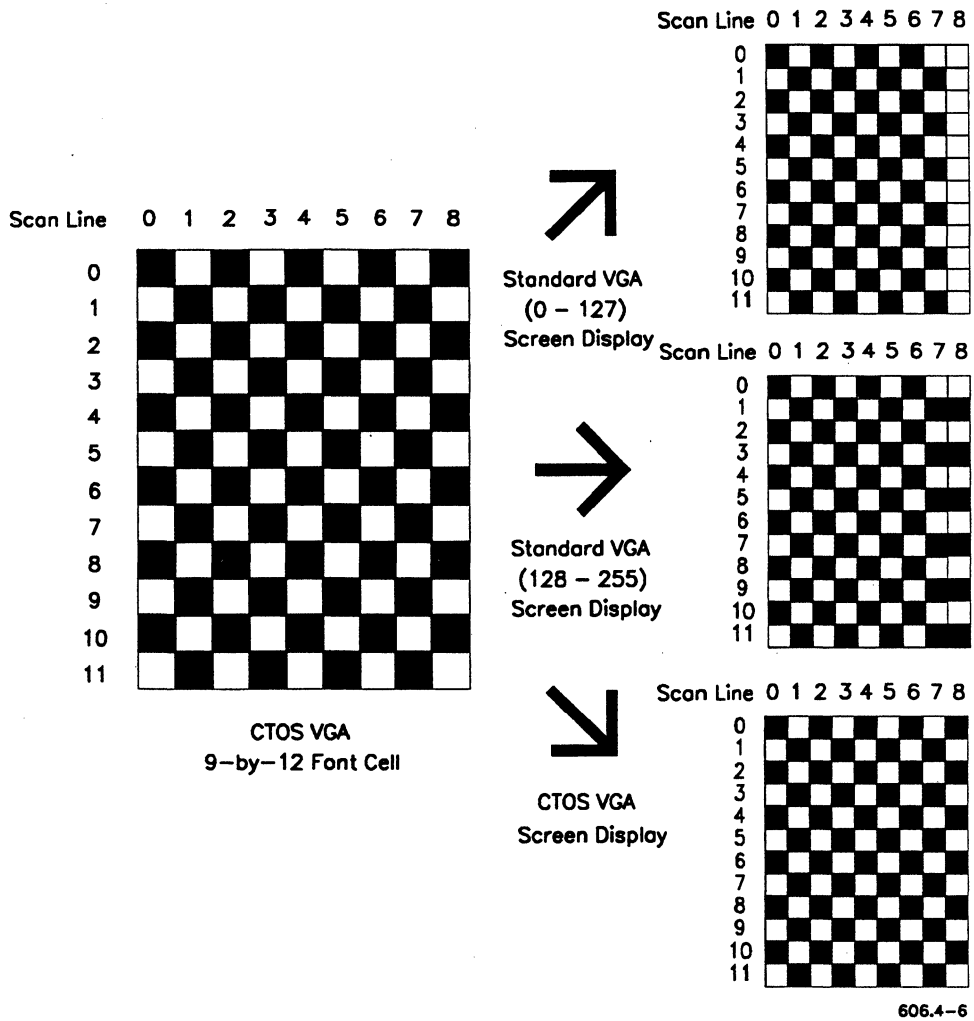


CTOS VGA Capability

With CTOS VGA, all nine bits are represented in the font and are supported by the hardware.

Figure 4-6 compares the Standard VGA and CTOS VGA representation of a "checkerboard" character. The character uses all 9 bits of the character cell width. Regardless of the character code, the Standard VGA representation distorts this character because information in the ninth bit of the CTOS VGA font is lost.

Figure 4-6. Representing 9-Bit-Wide Characters



Problem Description

As illustrated in Figures 4-4 and 4-5, the Standard VGA hardware generates the ninth bit of the font based on the character code. With CTOS VGA the ninth bit is actually represented as part of the font. Because of this implementation difference, the Standard VGA hardware cannot fully support CTOS VGA 9-bit-wide fonts.

Affected Applications

Applications that use the default system font are not affected by the difference in nine-bit-wide support. A new 8-by-16 system font file, *640x480Sys.font*, is packaged with CTOS III. This font is the default for Standard VGA hardware.

If, however, your application loads customized 9-by-12 fonts, it will be affected by the move to Standard VGA hardware. The extent to which it is affected depends upon the type of character represented in the font and the video display capability. For example, recall the 9-by-12 "checkerboard" character illustrated above. The Standard VGA hardware cannot represent this character in an 8-by-12 format without losing some information. As a further limitation, fonts loaded on systems with a flat panel display are adjusted automatically to a fixed number of scan lines for that hardware. (See "Display Resolutions" in this section.)

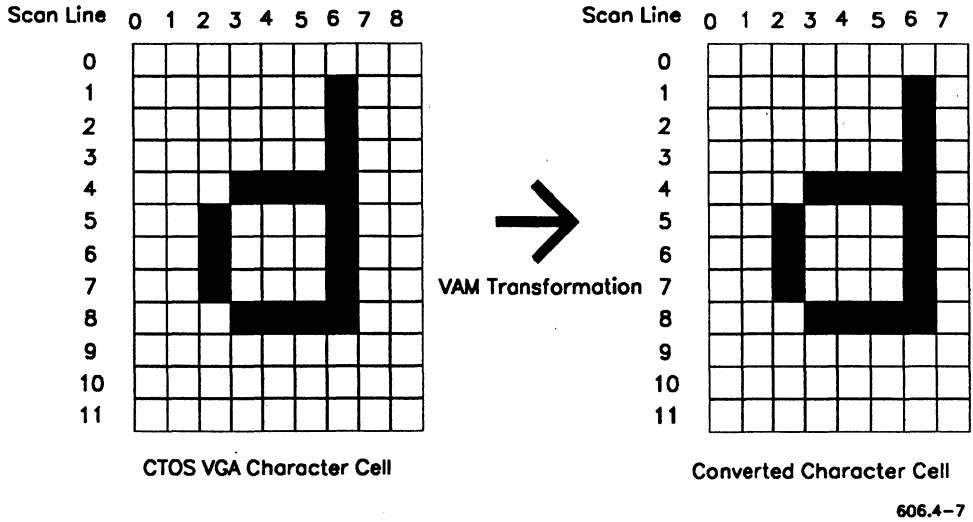
Recommended Solution

As a general rule, you should create 8-by-16 versions of existing 9-by-12 fonts, and that you modify your applications to load the 8-by-16 font when running on Standard VGA hardware. Use QueryVideo and QueryVidHdw to determine your system video functionality. Applications that rely on video functionality rather than video hardware type will be hardware independent.

As an interim solution, however, the CTOS Standard VGA VAM is enhanced to continue loading a CTOS VGA 9-by-12 font file. Before the 9-by-12 font is loaded into the display memory, VAM converts it to an 8-by-12 font.

Figure 4-7 shows a character cell before and after VAM performs the conversion process.

Figure 4-7. Converting to an 8x12 Font



To convert a 9-by-12 font to 8-by-12 before loading it into memory, VAM uses the following algorithm:

```
for each character cell do {
   UsePixels1-8 = false;
   for each scan line in the 9-by-12 font do {
       if (pixel 8 == 1)
           if (pixel 7 == 0) UsePixels1-8 = true;
       }
   if (UsePixels1-8) Use Pixels 1 through 8
   else Use Pixels 0 through 7
   }
```

Half-Bit Shift

Half-bit shift is the other CTOS VGA font-related feature not supported by Standard VGA. Half-bit shift is a hardware feature that shifts a character row to the right by one-half a pixel. This allows the font designer to create smoother appearing characters.

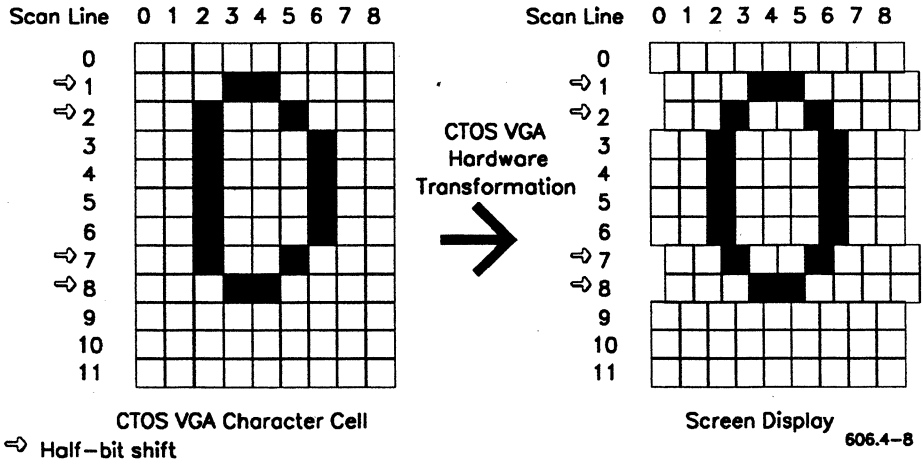
Standard VGA Capability

Standard VGA does not support half-bit shift.

CTOS VGA Capability

The CTOS VGA hardware does support half-bit shift. Figure 4-8 illustrates what a character cell looks like before and after the CTOS VGA hardware performs the half-bit shift function.

Figure 4-8. Half-Bit Shift



Problem Description

The Standard VGA hardware does not support half-bit shift. Any CTOS application that loads a customized font and takes advantage of this feature will be affected.

Affected Applications

Applications that use the default system font will not be affected. A new 8-by-16 system font file, *640x480Sys.font*, packaged with CTOS III is the default font file for Standard VGA hardware.

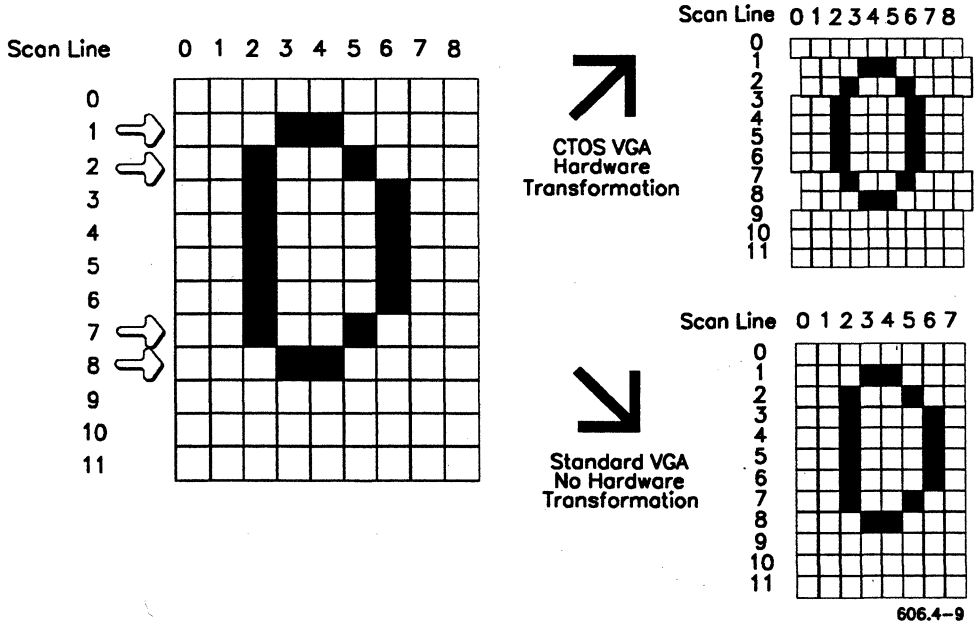
Applications that load a customized font and take advantage of the CTOS VGA half-bit shift capability will be affected. The extent to which an application is affected depends on the type of character represented in the font.

Recommended Solution

As a general rule, we recommend that you create 8-by-16 Standard VGA versions of existing 9-by-12 fonts, and that you modify your applications to load the 8-by-16 version when running on Standard VGA hardware. Use QueryVideo and QueryVidHdw to determine your system video functionality. Applications that rely on video functionality rather than video hardware type will be hardware independent.

As an interim solution, the CTOS Standard VGA VAM is enhanced to continue loading a CTOS VGA 9-by-12 font file. However, VAM performs no conversion to support half-bit shift. It simply ignores the attribute. As a result, certain characters may appear misshapen. Figure 4-9 illustrates the effect of ignoring the half-bit shift attribute on the "0" character.

Figure 4-9. Effects of Ignoring Half-Bit Shift



Display Resolution

The display resolution is another area in which CTOS VGA differs from Standard VGA. *Display resolution* takes into account the number of rows and columns in a video display and the character cell size. The ability to support the same set of display resolutions with each new release of the hardware has been a key value-add of the CTOS VGA hardware. With the move to Standard VGA, this feature can no longer be fully supported.

CTOS Standard VGA VAM does not support CTOS VGA display resolutions greater than 80 columns by 29 rows (the *enhanced* CTOS VGA display modes). CTOS applications that require an enhanced mode will require modification to run on Standard VGA hardware.

With the 4.1.5 release, CTOS Standard VGA VAM supports the two Standard VGA resolutions shown in Table 4-1. Except where limited by the hardware (see note below), CTOS Standard VGA VAM supports these resolutions on all systems using Standard VGA hardware.

Note: *Certain displays (for example, flat panel displays) support a fixed resolution only. On these systems, CTOS supports the hardware-supported resolution only.*

Table 4-1. Display Resolutions Supported by CTOS Standard VGA VAM

Resolution (Cols/Rows)	Character Cell (Pixels)
80x29	8-by-16
80x29	8-by-12

Standard VGA Capability

The Standard VGA hardware can be programmed to operate in a number of different display resolutions. To change the mode to a different display resolution, it is necessary to program multiple registers of the VGA chip in a consistent fashion. Details on the Standard VGA hardware display resolutions are beyond the scope of this section and are mentioned here only for the sake of comparison. CTOS VGA also runs in different resolutions, as noted below.

CTOS VGA Capability

CTOS VGA can be programmed to operate in a number of different display resolutions. Table 4-2 lists the supported resolutions.

Table 4-2. CTOS VGA Display Resolutions

Resolution (Cols/Rows)	Character Cell (Pixels)
80x29	9-by-12
80x34	9-by-10
132x29	7-by-12
132x34	7-by-10
80x38	12-by-20
146x38	7-by-20

Problem Description

None of the supported CTOS VGA display resolutions in Table 4-2 translates directly to a Standard VGA resolution. However, the 80x29 resolution with a 9x12 character cell (first entry in Table 4-2) is compatible on Standard VGA hardware. If your application loads a 9x12 font, CTOS Standard VGA VAM uses a character cell size of 8x12.

If you run your application on Standard VGA hardware and it requests any of the CTOS VGA resolutions other than 80x29, it will get back one resolution only: 80x29 with an 8x16 character cell. If the original resolution and character-cell size are very different, you can expect the resulting display to appear distorted.

Affected Applications

If your CTOS application requires a resolution other than 80x29, it is an affected application. To run your application on Standard VGA hardware, you will need to use one of the CTOS Standard VGA VAM-supported display resolutions listed in Table 4-1.

Recommended Solution

Modify applications that use unsupported display resolutions to use one listed in Table 4-1.

Character Attributes

The most noticable difference between CTOS VGA and Standard VGA lies in the area of character attributes.

CTOS VGA and Standard VGA are similar in that they both support 256 possible attribute combinations. However, CTOS VGA hardware supports attribute types unavailable on Standard VGA; likewise the Standard VGA hardware supports attribute types that CTOS VGA does not support.

Standard VGA Capability

Standard VGA supports a variety of different attribute configurations. Three possible options are shown in the figures below.

Figure 4-10 shows the first option. It allows the attribute byte to select one of sixteen colors for the foreground and one of sixteen colors for the background.

Figure 4-10. Standard VGA: Option One

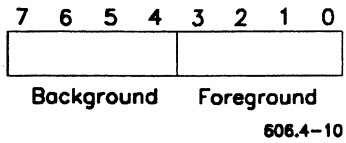


Figure 4-11 shows a second option. It allows the attribute byte to select one of sixteen colors for the foreground and one of eight colors for the background. In addition, the character may be configured to blink.

Figure 4-11. Standard VGA: Option Two

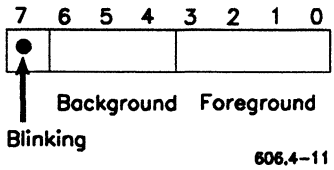


Figure 4-12 shows a third option. It allows the attribute byte to select one of eight foreground and one of eight background colors, and to select the character from an alternate font of 256 characters.

Figure 4-12. Standard VGA: Option Three

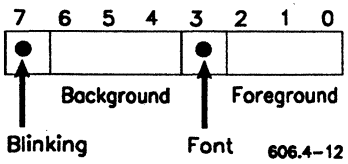


Table 4-3 lists the default colors loaded into the sixteen palette registers.

Table 4-3. Standard VGA Default Colors

Palette Register	Color
0	Black
1	Blue
2	Green
3	Cyan
4	Red
5	Magenta
6	Brown
7	White
8	Gray
9	Light Blue
10	Light Green
11	Light Cyan
12	Light Red
13	Light Magenta
14	Light brown (yellow)
15	Bright White

CTOS VGA Capability

On monochrome workstations or color workstations with color disabled, the attribute assigned to a character is defined as shown in Figure 4-13.

Figure 4-13. CTOS Attributes Without Color

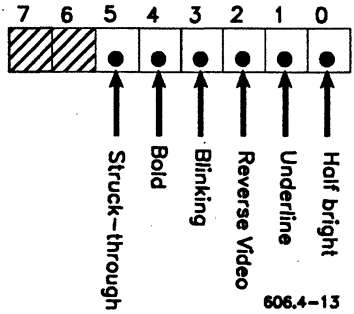


Figure 4-14 shows how the attributes are defined on a workstation with alphanumeric color enabled. (For details on using color, see the section entitled "Using Color" in the *CTOS Programming Guide*.)

Figure 4-14. CTOS Attributes With Alphanumeric Color

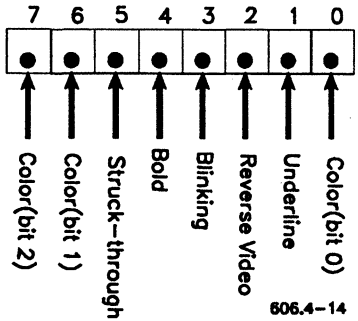


Table 4-4 lists the default colors loaded into the eight palette registers. (For details on the default colors, see the description of the SetAlphaColorDefault operation in the *CTOS Procedural Interface Reference Manual.*)

Table 4-4. CTOS VGA Default Colors

Palette Register	Foreground Color	Background Color
0	Green	Black
1	Half-green	Black
2	Cyan	Black
3	Half-cyan	Black
4	White	Black
5	Half-white	Black
6	Yellow	Black
7	Red	Black

Problem Description

CTOS attributes cannot be used to program Standard VGA hardware. For an existing CTOS application to display correctly on Standard VGA hardware, all CTOS attributes need to be mapped to Standard VGA attributes prior to being written to display memory.

Affected Applications

The following CTOS applications will be affected by the move to Standard VGA hardware:

- Applications that use combinations of CTOS attributes and colors that are incompatible with Standard VGA
- Applications that access the character map directly

The extent to which an application is affected will vary depending upon the application's use of attributes, color, and the character map.

Attributes

CTOS attributes and Standard VGA attributes are not equivalent. For example, Standard VGA hardware does not support a "struck-through" attribute. Since it is impossible to provide an exact mapping of CTOS attributes to Standard VGA attributes, an application's "look-and-feel" may change when running on Standard VGA hardware.

Color

On Standard VGA, it is necessary to use one of the color registers to represent the color black. Therefore, only seven (7) of the eight (8) CTOS colors can be displayed on the screen. The "look-and-feel" of applications that use the "missing" color will be affected.

Character Map

Because of the need to translate all attributes and the inability of VAM to quickly identify which attributes have changed, applications that directly access their character map may experience degraded performance. (For details, see "Accessing the Character Map," later in this section.)

Recommended Solution

As a general rule, we are recommending that you modify your applications to generate Standard VGA attributes when running them on Standard VGA hardware. In the interim, however, CTOS Standard VGA VAM is enhanced to map automatically CTOS attributes to Standard VGA attributes.

For applications that must provide a common "look-and-feel" on both Standard VGA and CTOS VGA hardware, you can use a common subset of attributes.

Mapping Attributes from CTOS to Standard VGA

CTOS Standard VGA VAM maps CTOS attributes to Standard VGA attributes.

An *attribute map* consists of 512 bytes. The first 256 bytes contain the mapping from CTOS attributes to Standard VGA attributes. The first byte of the map contains the mapping for CTOS attribute 0, the second byte contains the mapping for CTOS attribute 1, and so forth. The remaining 256 bytes are reserved for future use.

Attribute mapping is supported at the following levels:

- System-wide (default)
- Application-specific

You can configure an attribute map at either of these levels. Unless an application-specific map is defined, the system-wide attribute map is used. The system-wide default map is contained in the system video resource file, *VdmAttributes.res*.

To customize attribute mapping at either of these levels, you can choose from several Executive command line utilities. Four simple Executive commands allow you to specify alternatives to the defaults for color, bold, struck-through, or underlining. If you need to specify alternative displays for combinations of attributes, you can use the more powerful **VGA Character Attribute Mapper**. By using any of these utilities, you can change the defaults in the system-wide default map, or you can create an attribute map that you append to a specific application.

For details on how to use these utilities, see the *CTOS Executive Reference Manual.*

If you append an attribute map to your application, VAM uses this application-specific map instead of the system-wide map when the application is running.

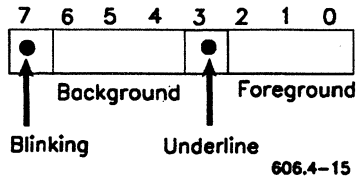
Note: *You cannot append an application-specific attribute map to a version 4 (real mode) run file. Real mode applications must use the system-wide attribute map.*

Default Attribute Mapping

This section describes the default mapping to be packaged with the CTOS Standard VGA VAM. (For details on how to customize the default mappings, see the *CTOS Executive Reference Manual.*

By default, Standard VGA hardware is programmed to support eight foreground colors, eight background colors, underlining, and blinking, as shown in Figure 4-15.

Figure 4-15. Standard VGA Default Attributes



Underlining is implemented using a Standard VGA alternate font.

Table 4-5 lists the default colors loaded into the eight palette registers. (For details on the default colors, see the description of the SetAlphaColorDefault operation in the *CTOS Procedural Interface Reference Manual.*) Because of the need to represent black, the default color map defines palette register 0 as black and makes half-cyan not available. (Compare the CTOS VGA color entries in Table 4-4 with the CTOS Standard VGA VAM colors shown in Table 4-5.) When processing a request to change the color palette, CTOS Standard VGA VAM "skips" the CTOS VGA palette register 3 shown in Table 4-4.

Table 4-5. CTOS Standard VGA VAM Default Colors

Palette Register	Color
0	Black
1	Green
2	Half-green
3	Cyan
4	White
5	Half-white
6	Yellow
7	Red

Tables 6 through 9 describe the 256 attribute combinations the above hardware setup will allow applications to access. These combinations are the available default values.

Table 4-6. Standard VGA Attributes (0 - F)

Std VGA Attribute	Foreground Color	Background Color	Blinking?	Underline?
0	Black(0)	Black(0)		
1	Green(1)	Black(0)		
2	Half-Green(2)	Black(0)		
3	Cyan(3)	Black(0)		
4	White(4)	Black(0)		
5	Half-white(5)	Black(0)		
6	Yellow(6)	Black(0)		
7	Red(7)	Black(0)		
8	Black(0)	Black(0)		yes
9	Green(1)	Black(0)		yes
A	Half-Green(2)	Black(0)		yes
B	Cyan(3)	Black(0)		yes
C	White(4)	Black(0)		yes
D	Half-white(5)	Black(0)		yes
E	Yellow(6)	Black(0)		yes
F	Red(7)	Black(0)		yes

Table 4-7. Standard VGA Attributes (10 - 1F)

Std VGA Attribute	Foreground Color	Background Color	Blinking?	Underline?
10	Black(0)	Green(1)		
11	Green(1)	Green(1)		
12	Half-Green(2)	Green(1)		
13	Cyan(3)	Green(1)		
14	White(4)	Green(1)		
15	Half-white(5)	Green(1)		
16	Yellow(6)	Green(1)		
17	Red(7)	Green(1)		
18	Black(0)	Green(1)		yes
19	Green(1)	Green(1)		yes
1A	Half-Green(2)	Green(1)		yes
1B	Cyan(3)	Green(1)		yes
1C	White(4)	Green(1)		yes
1D	Half-white(5)	Green(1)		yes
1E	Yellow(6)	Green(1)		yes
1F	Red(7)	Green(1)		yes

Table 4-8. Standard VGA Attributes (x0 - xF) (where x = (2,3,4,5,6,7))

Std VGA Attribute	Foreground Color	Background Color	Blinking?	Underline?
x0	Black(0)	Color(x)		
x1	Green(1)	Color(x)		
x2	Half-Green(2)	Color(x)		
x3	Cyan(3)	Color(x)		
x4	White(4)	Color(x)		
x5	Half-white(5)	Color(x)		
x6	Yellow(6)	Color(x)		
x7	Red(7)	Color(x)		
x8	Black(0)	Color(x)		yes
x9	Green(1)	Color(x)		yes
xA	Half-Green(2)	Color(x)		yes
xB	Cyan(3)	Color(x)		yes
xC	White(4)	Color(x)		yes
xD	Half-white(5)	Color(x)		yes
xE	Yellow(6)	Color(x)		yes
xF	Red(7)	Color(x)		yes

Table 4-9. Standard VGA Attributes (x0 - xF) (where x = (8,9,A,B,C,D,E,F))

Std VGA Attribute	Foreground Color	Background Color	Blinking?	Underline?
x0	Black(0)	Color(x)	yes	
x1	Green(1)	Color(x)	yes	
x2	Half-Green(2)	Color(x)	yes	
x3	Cyan(3)	Color(x)	yes	
x4	White(4)	Color(x)	yes	
x5	Half-white(5)	Color(x)	yes	
x6	Yellow(6)	Color(x)	yes	
x7	Red(7)	Color(x)	yes	
x8	Black(0)	Color(x)	yes	yes
x9	Green(1)	Color(x)	yes	yes
xA	Half-Green(2)	Color(x)	yes	yes
xB	Cyan(3)	Color(x)	yes	yes
xC	White(4)	Color(x)	yes	yes
xD	Half-white(5)	Color(x)	yes	yes
xE	Yellow(6)	Color(x)	yes	yes
xF	Red(7)	Color(x)	yes	yes

Default Attribute Mapping Tables

Tables 4-10 through 4-29 show the default attribute mapping packaged with CTOS Standard VGA VAM. The tables show all the attribute combinations supported on CTOS VGA hardware for monochrome and color workstations and what these combinations map to on Standard VGA hardware.

The CTOS VGA bold and struck-through attributes are ignored. Underlining, however, is fully supported. If you choose to display bold instead of underlining, you can accomplish this by using an Executive command line utility. (See the *CTOS Executive Reference Manual* for details.)

Default Attribute Mapping Without Alphanumeric Color

Tables 4-10 through 4-13 describe the default mapping for monochrome workstations or color workstations with color disabled.

Table 4-10. Monochrome Attribute Mapping (00 - 0F)

CTOS VGA	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
Half-bright		✓		✓		✓		✓		✓		✓		✓		✓
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Reverse Video					✓	✓	✓	✓					✓	✓	✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Bold																
Struck-through																

STD VGA																
Std VGA Attr	01	02	09	0A	10	20	18	28	81	82	89	8A	90	A0	98	A8
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Fore Color	01	02	01	02	00	00	00	00	01	02	01	02	00	00	00	00
Back Color	00	00	00	00	01	02	01	02	00	00	00	00	01	02	01	02

Table 4-11. Monochrome Attribute Mapping (10 - 1F)

CTOS VGA	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
Half-bright		✓		✓		✓		✓		✓		✓		✓		✓
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Reverse Video					✓	✓	✓	✓					✓	✓	✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Bold	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Struck-through																

STD VGA																
Std VGA Attr	01	02	09	0A	10	20	18	28	81	82	89	8A	90	A0	98	A8
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Fore Color	01	02	01	02	00	00	00	00	01	02	01	02	00	00	00	00
Back Color	00	00	00	00	01	02	01	02	00	00	00	00	01	02	01	02

Table 4-12. Monochrome Attribute Mapping (20 - 2F)

CTOS VGA	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
Half-bright		✓		✓		✓		✓		✓		✓		✓		✓
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Reverse Video					✓	✓	✓	✓					✓	✓	✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Bold																
Struck-through	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

STD VGA																
Std VGA Attr	01	02	09	0A	10	20	18	28	81	82	89	8A	90	A0	98	A8
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Fore Color	01	02	01	02	00	00	00	00	01	02	01	02	00	00	00	00
Back Color	00	00	00	00	01	02	01	02	00	00	00	00	01	02	01	02

Table 4-13. Monochrome Attribute Mapping (30 - 3F)

CTOS VGA	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F
Half-bright		✓		✓		✓		✓		✓		✓		✓		✓
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Reverse Video					✓	✓	✓	✓					✓	✓	✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Bold	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Struck-through	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

STD VGA																
Std VGA Attr	01	02	09	0A	10	20	18	28	81	82	89	8A	90	A0	98	A8
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Fore Color	01	02	01	02	00	00	00	00	01	02	01	02	00	00	00	00
Back Color	00	00	00	00	01	02	01	02	00	00	00	00	01	02	01	02

Default Attribute Mapping With Alphanumeric Color

Tables 4-14 through 4-29 describe the default mapping for workstations with alphanumeric color enabled.

Table 4-14. Color Attribute Mapping (00 - 0F)

CTOS VGA	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Reverse Video					✓	✓	✓	✓					✓	✓	✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Bold																
Struck-through																
Fore Color	00	01	00	01	00	01	00	01	00	01	00	01	00	01	00	01

STD VGA																
Std VGA Attr	01	02	09	0A	10	20	18	28	81	82	89	8A	90	A0	98	A8
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Fore Color	01	02	01	02	00	00	00	00	01	02	01	02	00	00	00	00
Back Color	00	00	00	00	01	02	01	02	00	00	00	00	01	02	01	02

Table 4-15. Color Attribute Mapping (10 - 1F)

CTOS VGA	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Reverse Video					✓	✓	✓	✓					✓	✓	✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Bold	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Struck-through																
Fore Color	00	01	00	01	00	01	00	01	00	01	00	01	00	01	00	01

STD VGA																
Std VGA Attr	01	02	09	0A	10	20	18	28	81	82	89	8A	90	A0	98	A8
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Fore Color	01	02	01	02	00	00	00	00	01	02	01	02	00	00	00	00
Back Color	00	00	00	00	01	02	01	02	00	00	00	00	01	02	01	02

Table 4-16. Color Attribute Mapping (20 - 2F)

CTOS VGA	20	21	22	23	24	25	26	27	28	29	2A	2B	2C	2D	2E	2F
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Reverse Video					✓	✓	✓	✓					✓	✓	✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Bold																
Struck-through	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Fore Color	00	01	00	01	00	01	00	01	00	01	00	01	00	01	00	01

STD VGA																
Std VGA Attr	01	02	09	0A	10	20	18	28	81	82	89	8A	90	A0	98	A8
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Fore Color	01	02	01	02	00	00	00	00	01	02	01	02	00	00	00	00
Back Color	00	00	00	00	01	02	01	02	00	00	00	00	01	02	01	02

Table 4-17. Color Attribute Mapping (30 - 3F)

CTOS VGA	30	31	32	33	34	35	36	37	38	39	3A	3B	3C	3D	3E	3F
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Reverse Video					✓	✓	✓	✓					✓	✓	✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Bold	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Struck-through	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Fore Color	00	01	00	01	00	01	00	01	00	01	00	01	00	01	00	01

STD VGA																
Std VGA Attr	01	02	09	0A	10	20	18	28	81	82	89	8A	90	A0	98	A8
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Fore Color	01	02	01	02	00	00	00	00	01	02	01	02	00	00	00	00
Back Color	00	00	00	00	01	02	01	02	00	00	00	00	01	02	01	02

Table 4-18. Color Attribute Mapping (40 - 4F)

CTOS VGA	40	41	42	43	44	45	46	47	48	49	4A	4B	4C	4D	4E	4F
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Reverse Video					✓	✓	✓	✓					✓	✓	✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Bold																
Struck-through																
Fore Color	02	03	02	03	02	03	02	03	02	03	02	03	02	03	02	03

STD VGA																
Std VGA Attr	03	03	0B	0B	30	30	38	38	83	83	8B	8B	B0	B0	B8	B8
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Fore Color	03	03	03	03	00	00	00	00	03	03	03	03	00	00	00	00
Back Color	00	00	00	00	03	03	03	03	00	00	00	00	03	03	03	03

Table 4-19. Color Attribute Mapping (50 - 5F)

CTOS VGA	50	51	52	53	54	55	56	57	58	59	5A	5B	5C	5D	5E	5F
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Reverse Video					✓	✓	✓	✓					✓	✓	✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Bold	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Struck-through																
Fore Color	02	03	02	03	02	03	02	03	02	03	02	03	02	03	02	03

STD VGA																
Std VGA Attr	03	03	0B	0B	30	30	38	38	83	83	8B	8B	B0	B0	B8	B8
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Fore Color	03	03	03	03	00	00	00	00	03	03	03	03	00	00	00	00
Back Color	00	00	00	00	03	03	03	03	00	00	00	00	03	03	03	03

Table 4-20. Color Attribute Mapping (60 - 6F)

CTOS VGA	60	61	62	63	64	65	66	67	68	69	6A	6B	6C	6D	6E	6F
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Reverse Video					✓	✓	✓	✓					✓	✓	✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Bold																
Struck-through	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Fore Color	02	03	02	03	02	03	02	03	02	03	02	03	02	03	02	03

STD VGA																
Std VGA Attr	03	03	0B	0B	30	30	38	38	83	83	8B	8B	B0	B0	B8	B8
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Fore Color	03	03	03	03	00	00	00	00	03	03	03	03	00	00	00	00
Back Color	00	00	00	00	03	03	03	03	00	00	00	00	03	03	03	03

Table 4-21. Color Attribute Mapping (70 - 7F)

CTOS VGA	70	71	72	73	74	75	76	77	78	79	7A	7B	7C	7D	7E	7F
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Reverse Video					✓	✓	✓	✓					✓	✓	✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Bold	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Struck-through	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Fore Color	02	03	02	03	02	03	02	03	02	03	02	03	02	03	02	03

STD VGA																
Std VGA Attr	03	03	0B	0B	30	30	38	38	83	83	8B	8B	B0	B0	B8	B8
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Fore Color	03	03	03	03	00	00	00	00	03	03	03	03	00	00	00	00
Back Color	00	00	00	00	03	03	03	03	00	00	00	00	03	03	03	03

Table 4-22. Color Attribute Mapping (80 - 8F)

CTOS VGA	80	81	82	83	84	85	86	87	88	89	8A	8B	8C	8D	8E	8F
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Reverse Video					✓	✓	✓	✓					✓	✓	✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Bold																
Struck-through																
Fore Color	04	05	04	05	04	05	04	05	04	05	04	05	04	05	04	05

STD VGA																
Std VGA Attr	04	05	0C	0D	40	50	48	58	84	85	8C	8D	C0	D0	C8	D8
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Fore Color	04	05	04	05	00	00	00	00	04	05	04	05	00	00	00	00
Back Color	00	00	00	00	04	05	04	05	00	00	00	00	04	05	04	05

Table 4-23. Color Attribute Mapping (90 - 9F)

CTOS VGA	90	91	92	93	94	95	96	97	98	99	9A	9B	9C	9D	9E	9F
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Reverse Video					✓	✓	✓	✓					✓	✓	✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Bold	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Struck-through																
Fore Color	04	05	04	05	04	05	04	05	04	05	04	05	04	05	04	05

STD VGA																
Std VGA Attr	04	05	0C	0D	40	50	48	58	84	85	8C	8D	C0	D0	C8	D8
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Fore Color	04	05	04	05	00	00	00	00	04	05	04	05	00	00	00	00
Back Color	00	00	00	00	04	05	04	05	00	00	00	00	04	05	04	05

Table 4-24. Color Attribute Mapping (A0 - AF)

CTOS VGA	A0	A1	A2	A3	A4	A5	A6	A7	A8	A9	AA	AB	AC	AD	AE	AF
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Reverse Video					✓	✓	✓	✓					✓	✓	✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Bold																
Struck-through	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Fore Color	04	05	04	05	04	05	04	05	04	05	04	05	04	05	04	05

STD VGA																
Std VGA Attr	04	05	0C	0D	40	50	48	58	84	85	8C	8D	C0	D0	C8	D8
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Fore Color	04	05	04	05	00	00	00	00	04	05	04	05	00	00	00	00
Back Color	00	00	00	00	04	05	04	05	00	00	00	00	04	05	04	05

Table 4-25. Color Attribute Mapping (B0 - BF)

CTOS VGA	B0	B1	B2	B3	B4	B5	B6	B7	B8	B9	BA	BB	BC	BD	BE	BF
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Reverse Video					✓	✓	✓	✓					✓	✓	✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Bold	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Struck-through	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Fore Color	04	05	04	05	04	05	04	05	04	05	04	05	04	05	04	05

STD VGA																
Std VGA Attr	04	05	0C	0D	40	50	48	58	84	85	8C	8D	C0	D0	C8	D8
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Fore Color	04	05	04	05	00	00	00	00	04	05	04	05	00	00	00	00
Back Color	00	00	00	00	04	05	04	05	00	00	00	00	04	05	04	05

Table 4-26. Color Attribute Mapping (C0 - CF)

CTOS VGA	C0	C1	C2	C3	C4	C5	C6	C7	C8	C9	CA	CB	CC	CD	CE	CF
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Reverse Video					✓	✓	✓	✓					✓	✓	✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Bold																
Struck-through																
Fore Color	06	07	06	07	06	07	06	07	06	07	06	07	06	07	06	07

STD VGA																
Std VGA Attr	06	07	0E	0F	60	70	68	78	86	87	8E	8F	E0	F0	E8	F8
Underline			✓	✓			✓	✓			✓	✓			✓	
Blinking									✓	✓	✓	✓	✓	✓	✓	
Fore Color	06	07	06	07	00	00	00	00	06	07	06	07	00	00	00	00
Back Color	00	00	00	00	06	07	06	07	00	00	00	00	06	07	06	07

Table 4-27. Color Attribute Mapping (D0 - DF)

CTOS VGA	D0	D1	D2	D3	D4	D5	D6	D7	D8	D9	DA	DB	DC	DD	DE	DF
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Reverse Video					✓	✓	✓	✓					✓	✓	✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Bold	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Struck-through																
Fore Color	06	07	06	07	06	07	06	07	06	07	06	07	06	07	06	07

STD VGA																
Std VGA Attr	06	07	0E	0F	60	70	68	78	86	87	8E	8F	E0	F0	E8	F8
Underline			✓	✓			✓	✓			✓	✓			✓	
Blinking									✓	✓	✓	✓	✓	✓	✓	
Fore Color	06	07	06	07	00	00	00	00	06	07	06	07	00	00	00	00
Back Color	00	00	00	00	06	07	06	07	00	00	00	00	06	07	06	07

Table 4-28. Color Attribute Mapping (E0 - EF)

CTOS VGA	E0	E1	E2	E3	E4	E5	E6	E7	E8	E9	EA	EB	EC	ED	EE	EF
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Reverse Video					✓	✓	✓	✓					✓	✓	✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Bold																
Struck-through	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Fore Color	06	07	06	07	06	07	06	07	06	07	06	07	06	07	06	07

STD VGA																
Std VGA Attr	06	07	0E	0F	60	70	68	78	86	87	8E	8F	E0	F0	E8	F8
Underline			✓	✓			✓	✓			✓	✓			✓	
Blinking									✓	✓	✓	✓	✓	✓	✓	
Fore Color	06	07	06	07	00	00	00	00	06	07	06	07	00	00	00	00
Back Color	00	00	00	00	06	07	06	07	00	00	00	00	06	07	06	07

Table 4-29. Color Attribute Mapping (F0 - FF)

CTOS VGA	F0	F1	F2	F3	F4	F5	F6	F7	F8	F9	FA	FB	FC	FD	FE	FF
Underline			✓	✓			✓	✓			✓	✓			✓	✓
Reverse Video					✓	✓	✓	✓					✓	✓	✓	✓
Blinking									✓	✓	✓	✓	✓	✓	✓	✓
Bold	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Struck-through	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Fore Color	06	07	06	07	06	07	06	07	06	07	06	07	06	07	06	07

STD VGA																
Std VGA Attr	06	07	0E	0F	60	70	68	78	86	87	8E	8F	E0	F0	E8	F8
Underline			✓	✓			✓	✓			✓	✓			✓	
Blinking									✓	✓	✓	✓	✓	✓	✓	
Fore Color	06	07	06	07	00	00	00	00	06	07	06	07	00	00	00	00
Back Color	00	00	00	00	06	07	06	07	00	00	00	00	06	07	06	07

Accessing the Character Map

CTOS video software supports different ways to access the video character map. The two basic categories of applications are listed below:

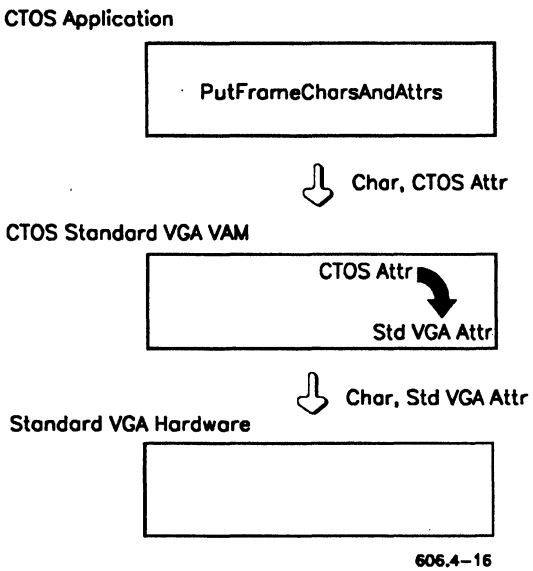
- Applications that access the character map indirectly by calling the operations listed in Table 4-30.
- Applications that directly access the character map by using the video structure *rgpVidMemLine*. These applications use the operations listed in Table 4-31 to lock and unlock the video.

To translate application-specified CTOS attributes into Standard VGA attributes, CTOS Standard VGA VAM must "filter" all attribute writes to the video device. (See Figure 4-16.) When applications use VAM operations, this attribute translation is accomplished with little or no performance impact.

If, however, an application accesses the character map directly, it may experience a performance degradation with attribute translation. In general, this type of application executes the following instruction sequence:

1. It locks the video.
2. It updates the character map.
3. It unlocks the video.

Figure 4-16. Mapping CTOS Attributes



On Standard VGA, CTOS applications are not allowed to write directly to video display memory. When an application unlocks the video, CTOS Standard VGA VAM copies the application's character map to the actual screen. During this copy procedure, CTOS Standard VGA VAM maps the CTOS attributes to Standard VGA attributes.

Protected mode applications directly accessing the video are not significantly affected. Features of the 386 hardware quickly identify those portion(s) of the character map that are changed due to attribute mapping.

Real mode applications, however, suffer a potentially severe performance hit if they frequently lock and unlock the video to make minimal screen changes. This situation would present itself, for example, if an application frequently locks the video, changes a single character, and then unlocks the video.

Table 4-30 lists the VAM operations an application can use to access the video device indirectly. Using these operations does not degrade performance. (For details on these operations, see the *CTOS Procedural Interface Reference Manual*.)

Table 4-30. Operations for Updating the Screen Indirectly

Operation Type Operation Name	Operation Name
Sequential access method (video byte stream) operations	
CheckpointBs	CloseByteStream
OpenByteStream	OutputToVid0
QueryVidBs	ReleaseByteStream
WriteBsRecord	WriteByte
Video access method operations	
FillFrame	FillFrameRectangle
FrameBackspace	MoveFrameRectangle
PosFrameCursor	PutCharsAndAttrs
PutFrameAttrs	PutFrameChars
PutFrameCharsAndAttrs	QueryBounds
QueryCharsAndAttrs	QueryCursor
QueryFrameBound	QueryFrameChar
QueryFrameCharsAndAttrs	QueryFrame
QueryFrameCursor	ScrollFrame
Video requests	
QueryFrameAttrs	QueryFrameString
Video display management operations	
QueryVidHdw	QueryVideo
LoadFontRam	ResetVideo
ResetFrame	InitVidFrame
SetScreenVidAttr	InitCharMap
SetVideoTimeOut	
Color programming operations	
ProgramColorMapper	SetAlphaColorDefault

Table 4-31 lists the operations an application uses to update the video device directly. Real-mode applications using these operations may experience severely degraded performance.

Table 4-31. Operations for Updating the Screen Directly

Operation Name	Operation Name
LockVideo	UnlockVideo
LockVideoForModify	UnlockVideoForModify

Section 5
Writing Partition-Managing Programs

Introduction

Certain kinds of applications need the ability to "spawn" other programs. For example, some applications allow users to run utility programs in the background, while the application continues to run in the foreground.

One way to accomplish this is for the application to create a child partition and run the utility program in that partition. Then, when the utility program has terminated, the application deallocates the child partition.

This section discusses the methods available to accomplish that.

For details on the relationship of partitions to the memory management styles of the different CTOS operating systems, see "Memory Management Style" in the *CTOS Operating Systems Concepts Manual.*

A Review of Partition Management Operations

As discussed in the *CTOS Operating System Concepts Manual*, the operating system provides several operations which aid in partition management. Context Manager also includes a library of operations that can be used when Context Manager is installed.

In general, if your program runs only when Context Manager is installed, use Context Manager operations. However, if your program needs to run whether or not Context Manager is installed, you must use the operating-system partition-management operations directly.

This section addresses the latter case, when a program must run whether or not Context Manager is present. See your Context Manager documentation for more information about the programmatic interface to Context Manager.

Creating a Partition and Loading a Program Into It

Creating partitions is not particularly complicated, as long as you perform the following steps in the correct order. The steps in creating a partition are:

1. Create the partition.
2. Swap its context into memory and set up the partition's environment.
3. Load a program into the partition.

In CTOS II, the newly created partition competes for memory with other partitions. The partition may be swapped out to make room for another partition. It's therefore necessary for the partition managing program to swap in or lock in the partition when the program is executed. In CTOS III, pages are swapped in as needed for program execution, and partition managing is not required. For details on swapping, see the "Partitions and Partition Management" section of the CTOS Operating System Concepts Manual.

The following sections discuss each of these steps in more detail. For an example of a complete partition-creating procedure, see the listings at the end of this section.

Creating a Partition

Example 5-1 shows a code fragment that creates a partition and swaps it into memory. In protected mode, any partition can create other partitions. In real mode operating systems, only the primary partition can create other partitions.

Example 5-1. Creating a Partition

```
erc = CreateBigPartition("MyChild", 7,(Quad)
      MemSize*64 /*parags*/, Par6, &userNumChild);
if(erc != ercOK)
  return(erc);
```

The fragment first creates a partition using CreateBigPartition. Using CreateBigPartition allows the program to create a partition of any size, while CreatePartition limits partition size to one megabyte. Note, however, that CreateBigPartition works only on protected-mode workstations.

The created partition is of type 6, which allows the partition to be resized dynamically by the operating system whenever a program is loaded into the partition. Also, note that the *MemSize* parameter represents a number of K bytes. This number must be multiplied by 64 to yield the paragraph count needed by the CreatePartition calls.

After creating a partition, the program can immediately load a program into the partition. Swapping the partition into memory and setting up the program's environment is required on CTOS II.

Partition-managing programs must be extremely careful about deallocating partitions they create. If a partition-managing program exits without deallocating a partition it created, the partition will remain intact, but will not be available to other programs. CTOS does not provide parent/child support for partitions.

Swapping the Partition Context Into Memory

After the program has created the partition, it swaps the new partition into memory. This causes the operating system to verify that enough physical memory is available and to allocate that memory to the newly created partition.

```
RetCleanup(SwapInContext(userNumChild));
```

RetCleanup is a macro that causes the program to deallocate the partition in case of error. See the complete program listing at the end of this section, which shows you how the macro is defined.

Setting Up the Program's Environment

After the partition has been created, the program may set up the partition environment for the program it plans to load. The amount of setup required depends on the resources that the program will use.

If the program uses only the virtual screen, application system control block, and variable length parameter block, the minimal setup consists of initializing only these things. This setup ensures that the program can write to its virtual screen, and can retrieve basic information about itself.

The code fragment in Example 5-2 shows the initialization of a newly created partition. Note that "Alt" requests are used for most of these programs.

Alt requests allow a program to send requests on behalf of another user number. For each request the operating system serves, there is also an Alt version. The only difference between a regular request and an Alt request is that the Alt requests take an additional parameter, the user number for whom the request should be sent.

Example 5-2. Initializing a New Partition's Environment

```
/* copy Vlpb and insert its address in the child partition's
Ascb */
RetCleanup(GetPStructure(lGetpVLPB, 0, &pVlpbMine));
RetCleanup( AltAllocMemoryLL(userNumChild,
              *(Word *)pVlpbMine, &pVlpbChild));
memcpy(pVlpbChild, pVlpbMine, *(Word *)pVlpbMine);
RetCleanup(GetPStructure(lGetpAscb, userNumChild,
                         &pAscbChild));
pAscbChild->pVlpb = pVlpbChild;

/* copy user name, signon password, and sys.cmds spec in
    new Ascb */
RetCleanup(GetPStructure(lGetpAscb, 0, &pAscbMine));
memcpy(pAscbChild->sbUserName, pAscbMine->sbUserName,
                  SCOPYAREA);
```

Example 5-2. Initializing a New Partition's Environment (cont.)

```
/* sneakily replace my pVlpb with childs so I can get
     params */
pTemp = pAscbMine->pVlpb;
pAscbMine->pVlpb = pVlpbChild;
/* get any parameters the user wants to enter */
RetCleanup(GetParams(pVlpbChild));
/* restore my Ascb */
pAscbMine->pVlpb = pTemp;

/* set child's exit run file */
RetCleanup( AltSetExitRunFile(userNumChild,
                "[Sys]<sys>Exec.run", 18,
                        NULL, 0, 0x80));
/* copy current path to new partition */
RetCleanup(GetUcb(&ucb, sizeof(ucb)));
RetCleanup( AltSetPath(userNumChild, ucb.rgbVol, ucb.cbVol,
              ucb.rgbDir, ucb.cbDir, ucb.rgbPswd, ucb.cbPswd));
if (ucb.cbPrefix)
    RetCleanup( AltSetPrefix(userNumChild, ucb.rgbPrefix,
                ucb.cbPrefix));
 if (ucb.cbNode)
    RetCleanup( AltSetNode(userNumChild, ucb.rgbNode,
                ucb.cbNode));

 /* initialize exec screen -- font doesn't need loading, nor
    double bar */
 RetCleanup( AltQueryVidHdw(userNumChild, &vidHdw,
             sizeof(vidHdw)));
 cLines = vidHdw.nLinesMax;
 cCols  = vidHdw.nColsNarrow;
 RetCleanup( AltResetVideo(userNumChild, cCols, cLines, TRUE,
             0, &sMap));
 RetCleanup( AltInitVidFrame(userNumChild, 1, 0, 0, cCols, 2,
             0, 0, 0, 0, 0));
 RetCleanup( AltInitVidFrame(userNumChild, 2, 0, 2, cCols, 1,
             0, 0, 0, 0, 0));
 RetCleanup( AltInitVidFrame(userNumChild, 0, 0, 3, cCols,
             (Byte)(cLines - 3), 0, 0, 0, 0, 0));
 RetCleanup( AltInitCharMap(userNumChild, 0, sMap));
 RetCleanup( AltSetScreenVidAttr(userNumChild, 1, 0xFF));
```

Loading the Program into the New Partition

Once the partition has been created, the partition-managing program can load a program into the new partition. This can happen whether or not the context has been swapped in and the environment has been set up. A program simply consists of a run file. Calling LoadInteractiveTask or LoadPrimaryTask causes the operating system to read the specified run file from disk into memory and then to create and schedule a process for the new program.

When LoadInteractiveTask or LoadPrimaryTask return to the caller, the newly loaded program is in memory and scheduled for execution. Example 5-3 shows a call to LoadInteractiveTask.

Example 5-3. Loading a Program for Execution

```
RetCleanup(LoadInteractiveTask(userNumChild,
  pRunFileName, strlen(pRunFileName), NULL, 0,
  StdPriority, FALSE));
RetCleanup(SwapInContext(userNumChild));
```

Finding the Program's Termination Status

As shown by the last three sections, creating a partition and executing a program in it is not particularly difficult. Removing a partition is equally simple, and is discussed later in this section. The difficult aspect of partition management is discovering when the program terminated and what its termination status was.

The Child-Termination Question

There is no operating system request that allows a program to determine the termination status of another program. A program can examine another program's ASCB to discover the most recent exit status code, but that provides only half the picture. How can it know whether the program in which it is interested has terminated? If that program is still active, the status code in the ASCB is irrelevant.

Also, the partition-managing program generally does not want to discover at some future time whether its child has finished executing. It wants to be notified immediately when the program in its child partition terminates.

A Solution: Defining a Termination Request

Because there is no predefined operating system request for it, there is no simple way to determine when a child terminates. It can be done, however, by defining your own termination request.

Whenever any program terminates, the operating system issues termination requests for it. These termination requests notify other programs (usually system services) that the program is terminating. This notification allows system services to perform any needed cleanup functions if the terminating program is a client.

If your partition-management program serves a termination request, it will be notified whenever a program on the workstation terminates. It can then identify whether the terminating program is its child and take appropriate action.

The partition managment program disables **ACTION+FINISH** and swapping (as a part of its initialization) and serves the termination request. The program creates a child and loads a program into it. When the management program receives a termination request, it knows that the child has terminated.

Calling the Termination Procedure

Example 5-4 shows a code fragment that calls the termination procedure named WaitForTermination.

Example 5-4. Calling a Child-Termination Procedure

```
erc = WaitForTermination(userNumChild);
if(erc == ercNoExitRunFile) {
  /* retrieve child's erc another way */
  erc = *(((Word *) pAscbChild) + 4);
  printf("\nChild partition exit run file not
    found or CM not installed.");
  }
```

Structure of the Termination Procedure

Example 5-5 shows an example of a child-termination procedure. This example does more than simply check for a single termination request. The procedure must do some extra work in case Context Manager software has been installed.

When Context Manager is installed, multiple applications run at the same time. It is possible (though unlikely) for two programs to terminate almost simultaneously. Because of this possibility, programs that serve termination requests may have more than one outstanding request in their request queues.

Therefore, the termination procedure must empty its request queue before it stops serving its termination request.

Example 5-5. Child-Termination Procedure

```
ErcType WaitForTermination(Word userNum) {
  RqTerminationType *pRqBlocRet;
  Word iMessageType;
  FlagType fAlive = TRUE;
  ErcType erc, ercRet = ercNoExitRunFile;
static Word MyExchange = 0;
  rqInfoType rqOldExchange;
  RetPreCleanup(DisableActionFinish(0xFF));
  RetPreCleanup(SetSwapDisable(0xFF));
  if(MyExchange == 0)
    RetPreCleanup(AllocExch(&MyExchange));
  RetPreCleanup(QueryRequestInfo(lMyTerminationRq,
                &rqOldExchange, sizeof(rqInfoType)) );
  RetPreCleanup(ServeRq(lMyTerminationRq, MyExchange));
  while(fAlive != FALSE) {
    RetCleanup(Wait(MyExchange, &pRqBlocRet));
    /* if is right user num and is termination request */
    if(pRqBlocRet->rqHead.userNum == userNum)  {
      pRqBlocRet->rqHead.ercRet = ercOK;
      fAlive = FALSE;
      ercRet = pRqBlocRet->ercTermination;
      }
    Respond(pRqBlocRet);
    }
```

Example 5-5. Child-Termination Procedure (cont.)

```
cleanup:
  while(erc != ercNoMessage) {
    /* make sure nobody's in queue */
    erc = Check(MyExchange, &pRqBlocRet);
    if(erc == ercOK)
      Respond(pRqBlocRet);
    }
  CheckErc( ServeRq(lMyTerminationRq, rqOldExchange.exch));

precleanup:
  CheckErc(SetSwapDisable(FALSE));
  CheckErc(DisableActionFinish(FALSE));
  return(ercRet);
}
```

Deallocating the Partition

Deallocating a partition is the simplest of the partition management operations. To deallocate a child partition, the parent program simply vacates the partition and then removes it. Example 5-6 shows a procedure that performs these operations.

Example 5-6. Partition Removal Procedure

```
void KillPartition(Word userNum) {
  ErcType MyErc;

  MyErc = VacatePartition(userNum);
  if(MyErc != ercOK && MyErc != ercPartitionVacant)
    CheckErc(MyErc);
  MyErc = RemovePartition(userNum);
  if(MyErc != ercOK)
    CheckErc(MyErc);
}
```

A Sample Partition-Management Program

Below is a complete listing of a simple partition-management program. A sample request definition file for the program's termination request follows the program listing.

Example 5-7. Simple Partition-Managing Program

```
/************************************
 * Filename:  Partitions.c
 * Date:      12/8/89
 * Author:    Alan Coleman
 * Compiler:  MicroSoft C
 * Modified:  Soyoung Ahn 05/10/93
 * This file spawns a partition, runs a program in it, then
 * kills the partition.
 ************************************/

#include <stdio.h>
#include <string.h>
#define AllocExch                                                    @
#define AssignKbd                                                    @
#define AssignVidOwner                                               @
#define Check                                                        @
#define CheckErc                                                     @
#define CreateBigPartition                                           @
#define Delay                                                        @
#define DisableActionFinish                                          @
#define ForwardRequest                                               @
#define GetPartitionStatus                                           @
#define GetPStructure                                                @
#define GetUcb                                                       @
#define GetUserNumber                                                @
#define LoadInteractiveTask                                          @
#define LoadPrimaryTask                                              @
#define QueryBigMemAvail                                             @
#define QueryRequestInfo                                             @
#define RgParamInit                                                  @
#define ServeRq                                                      @
#define SetSwapDisable                                               @
#define SwapInContext                                                @
```

Example 5-7. Simple Partition-Managing Program (cont.)

```
#define RemovePartition                                    @
#define Request                                            @
#define Respond                                            @
#define RgParamInit                                        @
#define RgParamSetSimple                                   @
#define VacatePartition                                    @
#define Wait                                               @

#define sdType
#define Syslit
#define UCBType
#define GetpStructureCase
#define RqHeaderType
#define rqInfoType

#include <ctos.lib.h>
#define KernelErc
#define mpErc
#define RxErc
#define RqErc
#include "erc.h"
#include "ctosTypes.h"

#define ParBackground 0
#define Par2          2
#define Par6          6
#define ParBatch      0xFF

#define StdPriority   0x80
#define lNotifyCMRq   258
#define lCMTerminationRq 273
#define lMyTerminationRq 0xE005

#define Go 0x1B


#define RetCleanup(x) if((erc = x) != ercOK) goto cleanup
#define RetPreCleanup(x) if((erc = x) != ercOK) goto
precleanup
```

Example 5-7. Simple Partition-Managing Program (cont.)

```
typedef struct {
  Word fhSwapFile;
  Pointer pVlpb;
  char unreferenced[58];
  char sbUserName[31];
  char sbPassword[13];
  char sbCmdFile[79];
} AscbType;

#define SCOPYAREA 31+13+79
/* size of sbUserName, sbPassword, sbCmdFile */

typedef struct {
  Byte level;
  Byte nLinesMax;
  Byte nColsNarrow;
  Byte nColsWide;
} VidHdwType;
typedef struct {
  RqHeaderType rqHead;
  ErcType      ercTermination;
  } RqTerminationType;
/* Just a status buffer */
Byte pStatus[250];
extern ErcType   _far _far _plm AltAllocMemoryLL(Word
                 userNum, Word cBytes,
                 Pointer ppSegmentRet);
extern ErcType   _far _plm AltInitCharMap(Word userNum,
                   Pointer pMap, Word sMap);
extern ErcType   _far _plm AltInitVidFrame(Word userNum,
                 Byte iFrame, Byte iColStart,
                 Byte iLineStart, Byte nCols, Byte nLines,
                 Byte borderDesc, Byte bBorderChar,
                 Byte bBorderAttr, Byte fDblHigh,
                 Byte fDblWide);
extern ErcType   _far _plm AltQueryVidHdw(Word userNum,
                 Pointer pBuffer, Word sBuffer);
extern ErcType   _far _plm AltResetVideo(Word userNum,
                 Byte nCols, Byte nLines, Byte fAttr,
                 Byte bSpace, Pointer psMapRet);
```

Example 5-7. Simple Partition-Managing Program (cont.)

```
extern ErcType    _far _plm AltSetNode(Word userNum,
                  Pointer pbNodeName, Word cbNodeName);
extern ErcType    _far _plm AltSetPath(Word userNum,
                  Pointer pbVolSpec, Word cbVolSpec,
                  Pointer pbDirName, Word cbDirName,
                  Pointer pbPassWord, Word cbPassWord);
extern ErcType    _far _plm AltSetPrefix(Word userNum,
                  Pointer pbPrefix, Word cbPrefix);
extern ErcType    _far _plm AltSetScreenVidAttr(Word userNum,
                  Word iAttr, Word fOn);
extern ErcType    _far _plm AltSetExitRunFile(Word userNum,
                  Pointer ExitRunFile, Word cbRunFile,
                  Pointer pbPassword, Word cbPassword,
                  Word Priority);

void KillPartition(Word userNum) {
  ErcType MyErc;
 MyErc = VacatePartition(userNum);
  if(MyErc != ercOK && MyErc != ercPartitionVacant)
    CheckErc(MyErc);

  MyErc = RemovePartition(userNum);
  if(MyErc != ercOK)
    CheckErc(MyErc);
}
/* NOTE:  This should be a separate process */
ErcType WaitForTermination(Word userNum) {
  RqTerminationType *pRqBlocRet;
  Word iMessageType;
  FlagType fAlive = TRUE;
  ErcType erc, ercRet = ercNoExitRunFile;
static Word MyExchange = 0;
  rqInfoType rqOldExchange;

  RetPreCleanup(DisableActionFinish(0xFF));
  RetPreCleanup(SetSwapDisable(0xFF));
  if(MyExchange == 0)
    RetPreCleanup(AllocExch(&MyExchange));
```

Example 5-7. Simple Partition-Managing Program (cont.)

```
    RetPreCleanup(QueryRequestInfo(lMyTerminationRq,
                  &rqOldExchange,
                      sizeof(rqInfoType)) );
    RetPreCleanup(ServeRq(lMyTerminationRq, MyExchange));

    while(fAlive != FALSE) {
      RetCleanup(Wait(MyExchange, &pRqBlocRet));
      /* if is right user num and is termination request */
      if(pRqBlocRet->rqHead.userNum == userNum)  {
        pRqBlocRet->rqHead.ercRet = ercOK;
        fAlive = FALSE;
        ercRet = pRqBlocRet->ercTermination;
        }
      Respond(pRqBlocRet);
      }


cleanup:
    while(erc != ercNoMessage) {
      /* make sure nobody's in queue */
      erc = Check(MyExchange, &pRqBlocRet);
      if(erc == ercOK)
        Respond(pRqBlocRet);
      }

    CheckErc( ServeRq(lMyTerminationRq, rqOldExchange.exch));
  precleanup:
    CheckErc(SetSwapDisable(FALSE));
    CheckErc(DisableActionFinish(FALSE));

    return(ercRet);
  }

  ErcType GetParams(Pointer pVlpb) {
  #define MAXPARAMS 8
    ErcType erc;
    int     iCurrent = 1;
    sdType  sdParam;
    char    rgbString[80];
    sdParam.pb = rgbString;
```

Example 5-7. Simple Partition-Managing Program (cont.)

```
  sdParam.cb = 0;
  erc = RgParamInit(pVlpb, 0, MAXPARAMS);
  printf("\nType param or press Go, then Return to
          execute.\n");
  while(TRUE) {
    printf("Enter parameter %d:  ", iCurrent);
    gets(rgbString);
    if(rgbString[0] == Go)
      break;
    sdParam.cb = strlen(rgbString);
    erc = RgParamSetSimple(iCurrent, ((Pointer) &sdParam));
    iCurrent++;
    }
  return(erc);
}

ErcType RunSubTask(char *pRunFileName, Word MemSize) {
  ErcType erc, ercRet=0;
  Quad   dMemAvail;
  Word    userNumChild, userNum;  AscbType *pAscbChild,
          *pAscbMine;
  Pointer pVlpbMine, pVlpbChild;
  Pointer pTemp;
  UCBType ucb;
  VidHdwType vidHdw;
  Byte/*Word*/ sMap, cLines, cCols;

  erc = CreateBigPartition("MyChild", 7,(Quad) MemSize*64
                                    Par6, &userNumChild);
  if(erc != ercOK)
    return(erc);

  RetCleanup(SwapInContext(userNumChild));
```

Example 5-7. Simple Partition-Managing Program (cont.)

```
/* copy Vlpb and insert its address in the child
   partition's Ascb */
RetCleanup(GetPStructure(lGetpVLPB, 0, &pVlpbMine));
RetCleanup( AltAllocMemoryLL(userNumChild,
            *(Word *)pVlpbMine, &pVlpbChild));
memcpy(pVlpbChild, pVlpbMine, *(Word *)pVlpbMine);
RetCleanup(GetPStructure(lGetpAscb, userNumChild,
           &pAscbChild));
pAscbChild->pVlpb = pVlpbChild;

/* copy user name, signon password, and sys.cmds spec in
   new Ascb */
RetCleanup(GetPStructure(lGetpAscb, 0, &pAscbMine));
memcpy(pAscbChild->sbUserName, pAscbMine->sbUserName,
       SCOPYAREA);

/* replace my pVlpb with childs so I can get params */
pTemp = pAscbMine->pVlpb;
pAscbMine->pVlpb = pVlpbChild;
/* get any parameters the user wants to enter */
RetCleanup(GetParams(pVlpbChild));
/* restore my Ascb */
pAscbMine->pVlpb = pTemp;

/* set child's exit run file */
RetCleanup( AltSetExitRunFile(userNumChild,
            "[Sys]<sys>Exec.run", 18, NULL, 0, 0x80));

/* copy current path to new partition */
RetCleanup(GetUcb(&ucb, sizeof(ucb)));
RetCleanup( AltSetPath(userNumChild, ucb.rgbVol,
            ucb.cbVol, ucb.rgbDir, ucb.cbDir, ucb.rgbPswd,
            ucb.cbPswd));
if (ucb.cbPrefix)
  RetCleanup( AltSetPrefix(userNumChild, ucb.rgbPrefix,
              ucb.cbPrefix));
if (ucb.cbNode)
  RetCleanup( AltSetNode(userNumChild, ucb.rgbNode,
              ucb.cbNode));
```

Example 5-7. Simple Partition-Managing Program (cont.)

```
  /* initialize exec screen -- font doesn't need loading,
     nor double bar */
  RetCleanup( AltQueryVidHdw(userNumChild, &vidHdw,
              sizeof(vidHdw)));
  cLines = vidHdw.nLinesMax;
  cCols  = vidHdw.nColsNarrow;
  RetCleanup( AltResetVideo(userNumChild, cCols, cLines,
              TRUE, 0, &sMap));
  RetCleanup( AltInitVidFrame(userNumChild, 1, 0, 0, cCols,
              2, 0, 0, 0, 0, 0));

  RetCleanup( AltInitVidFrame(userNumChild, 2, 0, 2, cCols,
              1, 0, 0, 0, 0, 0));
  RetCleanup( AltInitVidFrame(userNumChild, 0, 0, 3, cCols,
              (Byte)(cLines - 3), 0, 0, 0, 0, 0));
  RetCleanup( AltInitCharMap(userNumChild, 0, sMap));
  RetCleanup( AltSetScreenVidAttr(userNumChild, 1, 0xFF));
  RetCleanup(LoadInteractiveTask(userNumChild, pRunFileName,
      strlen(pRunFileName), NULL, 0, StdPriority, FALSE));

  RetCleanup(SwapInContext(userNumChild));


  erc = WaitForTermination(userNumChild);
  if(erc == ercNoExitRunFile) {
    /* retrieve child's erc another way */
    erc = *(((Word *) pAscbChild) + 4);
    printf("\nChild partition exit run file not found or
                    CM not installed.");
    }

cleanup:
  KillPartition(userNumChild);
  return(erc);
}


int main() {
char rgbRunFile[80];
char rgbMemSize[20];
Word MemSize;
ErcType ercRet;
```

Example 5-7. Simple Partition-Managing Program (cont.)

```
  while(TRUE) {
    printf("\nEnter run file name:  ");
    gets(rgbRunFile);
    if(rgbRunFile[0] == 0)
      break;
    printf("How much memory does it need (in K):  ");
    gets(rgbMemSize);
    MemSize = atoi(rgbMemSize);

    ercRet = RunSubTask(rgbRunFile, MemSize);
    printf("\nExecuted %s.  Returned erc = %d\n",
            rgbRunFile, ercRet);
    }
  return(0);
}
```

Example 5-8. Termination Request Definiton File

```
:WsAbortRq:      0E005h
:TerminationRq:  0E005h

:RequestCode:    0E005h
:RequestName:    TerminationAndWsAbortRq
:Version:        1
:LclSvcCode:     0002h
:ServiceExch:    exchInstalledMastr
:sCntInfo:       6
:nReqPbCb:       0
:nRespPbcb:      0
:Params:         none
:NetRouting:     noRouting
:SrpRouting:     rLocal
```

Section 6
Software Installation: The Installation Manager

Introduction

If you have written an application or system service, users need to install it as a software product on the hard disk of their client workstation or server. You need to create several files to accomplish this process. This section describes the installation process and provides guidelines for writing the installation files as well as instructions on how to place these files on the installation media. This section assumes that you are familiar with the batch processing of job control language (JCL) files. If you are not, refer to the *CTOS Batch Manager II Installation, Configuration, and Programming Guide* before proceeding.

Two run files, two message files, and a command file provided with Standard Software, Version 12.3, work together to install software. These files are *InstallMgr.run, InstallMgrMsg.bin, Batch.run, BatchMsg.bin*, and *Install>English.cmds*.

InstallMgr.run initiates installation and deinstallation of software. It does the preliminary work needed to accomplish the installation, such as collecting installation parameter data from the user configuration file or interactively from the user, verifying the availability of resources required by the product, and copying installation files to a temporary location on the hard disk where their contents can be accessed by *Batch.run*. *Install>English.cmds* contains commands commonly used in installation scripts. Batch uses this file to resolve $Command statements in the installation scripts.

Once the installation environment is set up, the Installation Manager passes control to Batch. Batch does the actual work of installing your software product. Guided by information provided by the Installation Manager, Batch performs the job steps in the installation script. These steps typically include copying the files for your product to the hard disk and merging your product's commands with the proper command file. For your understanding of the installation process, this section begins by describing the Installation Manager and Batch programs in greater detail. Following the program descriptions, this section describes the contents of the installation files, explains how you create these files, and instructs you on how to organize the content of your distribution media. Later in this section, example scripts show you how to write the instructions Batch needs to carry out the installation.

Key Concepts

This section describes key concepts related to the installation process: the type of installation (floppy, tape, or server), whether the installation is public or private, and whether the product needs to be divided into subpackages for separate installation.

Types of Installation: Floppy, Tape, Server

The *installation media* is the type of media on which your product is sold to the customer. It could be 5.25-inch or 3.5-inch disks or Quarter-Inch Cartridge (QIC) tape. A *floppy installation* is an installation whose media is floppy disks. A *tape installation* is an installation whose media is QIC tape.

A *server installation* is an installation whose media is a disk on the server. A subpackage must be installed *publicly* before it can be installed from the server.

Another software installation method that allows you to install software remotely to large numbers of servers and workstations requires the SWD package. See the SWD documentation that accompanies the SWD product.

Public vs. Private Installation

A *public* installation is a floppy or tape installation that puts files onto the server and the commands into the file *[!Sys]<Sys>Cluster.cmds*. This makes the subpackage available to all members of the cluster and provides two practical benefits. First, the subpackage can be installed on a workstation from the server instead of being installed from floppy disks or tape. Second, the workstation can access seldom used subpackage commands stored on the server instead of taking up workstation disk space. You may need to include optional fields in the control file (see below) when a subpackage is installed publicly. Subpackages should be publically installable.

A *private* installation puts files onto a local hard disk on the workstation from which the Installation Manager is invoked. The commands are put into *[Sys]<Sys>Sys.cmds*.

Subpackages

A *product* is the set of files that make up the entity you are distributing. It consists of system services, (client) run files, and supporting files (for example, configuration files and request files) required by your service and/or run files. These are files that you create and distribute. Your product does not include files like Context Manager's configuration file, which is part of Context Manager product, nor *Request.sys*, which is part of the Standard Software product.

Before putting your product onto distribution media, you must decide whether it is atomic or can be divided into smaller pieces. This is determined most of the time by answering the question: "Are there parts of the product that some users would want and others would not?" For example, Standard Software consists of more than 15 pieces (subpackages). On a smaller scale, a product might consist of a system service (that would only be installed on the server) and several client programs. In this case, you would divide it into two subpackages. Using subpackages allows you to take full advantage of the variables and other resources that the Installation Manager provides to you as script author. By using subpackages, you ask the user to select subpackages instead of individual files.

A *subpackage* is a part of an application that is self-sufficient. It will have its own entry in the installation database. For instance, an accounting product might have four subpackages: Accounts Payable, Accounts Receivable, Payroll, and General Ledger.

The *installation database* is the file where the Installation Manager keeps the information about the subpackage. The Installation Manager gets the information from the control file, described later in this section. Subpackages that are installed publicly are put into *[!Sys]<Installed>Cluster.installed*. Subpackages that are installed privately are put into *[Sys]<Installed>Sys.installed*.

A *package* is a special case where the product consists of one subpackage.

The Installation Manager

To install a software product, a user invokes the Installation Manager using the Executive **Installation Manager** command. Information on using this command is contained in the *CTOS System Administration Guide*. The Installation Manager queries for the type of installation the user wants to perform and then sets up the appropriate environment.

A user can also specify directly which type of installation is needed through use of the **Floppy Install, Tape Install**, or **Server Install** commands.

When invoked by the **Installation Manager** command, the Installation Manager performs the following procedure sequence. When invoked by the **Floppy Install, Tape Install**, or **Server Install** commands, the sequence begins at step 3.

1. It allows the user to choose a function: *Install New Software, Show Installed Software*, or *Remove Installed Software*. We will assume the user chooses the *Install New Software* option.
2. It queries the user for the source of the installation, such as floppy disk, QIC tape, or the server.
3. It sets the installation variables based on parameter values obtained from the user configuration file or from the user choosing the *Examine / Change Defaults* option from the Installation Defaults menu. (All the installation variables are described in "Installation Variables," later in this section.)

4. It then asks the user to choose the subpackage(s) she or he wishes to install. If the installation is performed from the server, only one subpackage may be chosen at a time.
5. It calculates whether disk space is sufficient for installing the product using data obtained from the installation control file (described later).
6. The Installation Manager then passes control to Batch and provides it with the values of the installation variables.
7. Batch processes the installation script and then returns control to the Installation Manager.
8. If the installation succeeds, the database is updated to reflect the installation of the subpackage.

If the installation type was from floppy and the user selected more than one subpackage, steps 6 through 8 are repeated for each subpackage.

If the installation type was from server, then the user begins again at step 1. Note that step 3 only has to be done once each time the Installation Manager is invoked. The Installation Manager does not set or reset these values except upon initial invocation and during step 3.

There is only one installation script for tape installation, regardless of the number of subpackages that the user selects. Therefore steps 6 and 7 are only done once, but step 8 is performed once for each subpackage selected.

If a previous installation failed, the Installation Manager allows the user to restart the installation instead of starting over. The *[Sys]<Installed>Install.log* provides the installation details in the event of a failure. See the subsection "Restarting an Installation," later in this section.

Batch

Once the Installation Manager passes control to Batch, Batch does the actual work of installing your subpackage on the hard disk. Using the values in the installation variables, Batch executes the statements in your installation script.

Instructions in the installation script typically include:

- Copying the files for your subpackage from the installation media to the appropriate destination on the hard disk
- Merging your requests into the user's request file
- Merging your commands into the appropriate command file
- Creating directories on the hard disk

See the end of the section for examples of installation scripts

Installation Files

You create four installation files as part of your subpackage:

- A control file (required), *Install.ctrl*
- An installation script file (required), *Install.jcl*
- A message file (optional), *InstallMsg.bin*
- A command file (optional), *Install.cmds*

The Installation Manger reads these files from the installation media and copies them to a temporary location on the hard disk where their contents are available for the duration of the installation process. The first two files are ASCII files and are created by using the Editor. The message file is created from an ASCII file using the Executive command **Create Message File**. The command file is a typical command file and is created by the Command File Editor.

Control File

The *control file* contains information needed to guide the installation process. The Installation Manager reads the contents of this file and uses the information it contains to perform functions, such as

- Displaying the package name and version number on the user screen
- Verifying that there are adequate space and resources for the installation
- Updating the database

All information in the control file is placed in the user's installation database upon successful installation.

The control file for Mouse Services, for example, contains the following fields and parameter entries:

```
:PackageName:Mouse
:Version:12.3.0
:RequiredDiskSpace:381
:Verify:
:Directory:<Sys>
:Commands: 'Install Mouse Service' 'Deinstall Mouse Service'
'Set Mouse Controls'
:ProductFiles: 1024x768_PntSys.icon 720x348_PntSys.icon
DefaultCursor.icon Mouse.run MouseBm.run MouseCh.run
MouseForm.lib MouseMsg.bin SetMouse.run
```

Values for each keyword start on the same line as the keyword. Spaces may exist between the ending colon of the keyword and the beginning character of the value.

The value includes one or more elements. In other words, a value can be a list of elements.

If the value is a list of different elements, separate the elements with a space, not a comma. Place single quotation marks around individual elements if they contain embedded spaces.

The value is terminated by a carriage return. This is especially important to remember when dealing with values such as *:ProductFiles:* which contain a list of files that might wrap around several lines. **Do not** use carriage returns to make the file look pretty. A carriage return *terminates* the list.

The contents of this file are of the form *:Keyword: Value* as shown in the previous example.

Of the keywords listed next, only *:PackageName:* and *:Version:* are required.

Control File Keywords

:PackageName: (required)

The *:PackageName:* field should contain a name that clearly identifies the software product to the user. The maximum length is 30 characters. The combined length of this field and the *:Version:* field cannot be greater than 34 characters. Embedded spaces are allowed. This name must be unique. "OFIS Designer," for example, would be easily identified by someone installing the package from a server. (Note that the field name technically should be *:SubPackageName:*, but remains *:PackageName:* so that older installations do not break.)

:MultiPkgName: (optional)

The *:MultiPkgName:* specifies the name of a subpackage. See *:PackageName:* above for restrictions. Each subpackage must have a *:MultiPkgName* entry associated with it. If this entry appears in a control file, the control file must contain exactly one *:PackageName:* entry and between two and fifteen *:MultiPkgName:* entries. Everything else is ignored.

:Version: (required)

The *:Version:* field lets the Installation Manager know if the same or an earlier version of the product is already installed or if the installation is being performed for the first time. The maximum length is 15 characters. The combined length of this field and the *:PackageName:* field cannot be greater than 34 characters. Embedded spaces are not allowed.

:RequiredDiskSpace: (optional)

The *:RequiredDiskSpace:* field specifies the total length in K Bytes of all installation files in your subpackage. Although the units are K Bytes, do not append K to the number. (See also *:ProductFiles:* below.)

:PublicRequiredDiskSpace: (optional)

The *:PublicRequiredDiskSpace:* field is the total size of all the files in your subpackage when it is installed publicly. See *:RequiredDiskSpace:* for restrictions. If there is no difference in a public and private installation, do not include this entry. (See also *:PublicProductFiles:* below.)

:Verify (optional)

The *:Verify:* keyword, if included, indicates that the Installation Manager should verify the time/date stamp during installation. This entry should always be included since the verify feature is a useful function of the Installation Manager. If only a portion of the product files is to be installed, dividing the product into appropriate subpackages allows the user to install an entire subpackage and to verify the time and date of each product file.

:Directory: (optional)

The *:Directory:* field specifies the directory in which the files will reside. Include angle brackets. Usually the value is <*Sys*>. The maximum length is 14 characters. If you use this entry, do not put the directory in the file specifications given in the *:ProductFiles:* and *:OverwriteNoFiles:* fields. If you must place files in more than one directory and cannot use subpackages, do not use this entry. Instead, place the directories in the file specifications given in the *:ProductFiles:* and *:OverwriteNoFiles:* fields.

:Commands: (optional)

The *:Commands:* field contains a list of all commands used to invoke the run files and/or services in the subpackage. These commands are removed from the command file when the user removes the subpackage.

:ProductFiles: (optional)

The *:ProductFiles:* field includes a comprehensive list of files that make up your subpackage. They are deleted when the subpackage is deinstalled. When the user requests a backup of the previous version of software (the next time you release your product), these files along with those listed in the *:RelatedFiles:* field, will be archived. The length of these files should be added and that number placed in the *:RequiredDiskSpace:* entry. Do not include volume information in the file name. The user controls the volume name. Do not include directory information if you included the *:Directory:* field. The directory name will be taken from the *:Directory:* field. The product file names can include spaces if the name is enclosed in single quotation marks.

If the verify feature is enabled, after returning from Batch, the Installation Manager checks the files in the *:ProductFiles:* list to see whether their modification/creation date-time stamps are greater than when the Installation Manager was invoked. If they are not, the installation fails, and a message with the file name of the nonupdated file is put in the installation log file.

:PublicProductFiles: (optional)

The *:PublicProductFiles:* field contains a list of files that make up your subpackage when it is installed publicly. If there is no difference in a public and private installation, do not include this entry. If some files are common to both public and private installations, make sure you include them in both the *:PublicProductFiles:* and *:ProductFiles:* fields. The product file names can include spaces if the name is enclosed in single quotation marks.

If the verify feature is enabled, after returning from Batch, the Installation Manager checks the files in the *:PublicProductFiles:* list to see whether their modification/creation date-time stamps are greater than the time that the Installation Manager was invoked. If they are not, the installation fails, and a message with the file name of the nonupdated file is put in the installation log file.

:OverwriteNoFiles: (optional)

The *:OverwriteNoFiles:* field specifies a list of file names that belong to your subpackage, but which should not be overwritten by the installation process if they already exist. Do not include volume information in the file name. The user controls the volume name. Do not include directory information if you included the *:Directory:* field. The directory name is taken from the *:Directory:* field. User-configurable files would go into this list-for example, *Queue.index*. These names must also be in the *:ProductFiles:* list.

If the verify feature is enabled, after returning from Batch, the Installation Manager checks the files in the *:OverwriteNoFiles:* list to see whether they exist. If they don't, the installation fails, and a message with the file name of the nonexistent file is put in the installation log file. Remember to include these file names in the *:ProductFiles:* list.

:PublicOverwriteNoFiles: (optional)

The *:PublicOverwriteNoFiles:* field contains a list of files that belong to your subpackage, but which should not be overwritten by a public installation if they already exist. See *:OverwriteNoFiles:* for restrictions. The names here must also be in the *:PublicProductFiles:* list. If there is no difference in a public and private installation, do not include this entry.

If the verify feature is enabled, after returning from Batch, the Installation Manager checks the files in the *:PublicOverwriteNoFiles:* list to see whether they exist. If they don't, the installation fails, and a message with the file name of the nonexistent file is put in the installation log file. Remember to include these file names in the *:PublicProductFiles:* list.

:RelatedFiles: (optional)

The *:RelatedFiles:* field is a list of files that the installation process updates but that are not part of your product, for example, *:Request.sys:*. At this time, there is no way to specify files such as the *.user* file and the Context Manager configuration file, since the names can be changed by the user. at install time. These files are archived along with the files listed in the *:ProductFiles:* entry. The files listed in this field can include *[Volume]<Directory>* specifications.

:RequestCodes: (optional)

The *:RequestCodes:* field is a list of request codes for which your subpackage issues ServeRq calls. The codes must be separated by a space, not a comma. The request codes can be either decimal or hexadecimal numbers. If they are hexadecimal, append h, for example, 0E001h. To add request codes, your jcl scripts must use the List Request Set, Make Request Set, and Update Request Set jcl commands which are discussed later in this section. These request codes are removed when your subpackage is deinstalled.

:RequiredMemory: (optional)

The *:RequiredMemory:* field specifies the amount of memory that your largest run file requires. This entry is not currently used.

:NoDbUpdate: (optional)

The *:NoDbUpdate:* keyword, if included, indicates that the Installation Manager should not update the installation database. Normally this field would not be included in the control file. However, some remote installations (via SWD, a different CTOS installation method) may want to copy files only or perform some other functions and therefore don't want the database updated.

:PermanentFiles: (optional)

The *:PermanentFiles:* field, if included, specifies a list of file names that belong to your subpackage that should not be deleted if the subpackage is deinstalled. Do not include directory information if you included the *:Directory:* field. The directory name will be taken from the *:Directory:* field. These file names must also be in the *:ProductFiles:* or *:PublicProductFiles:* list.

:PreInstallFiles: (optional)

The *:PreInstallFiles:* field, if included, specifies a list of file names that belong to your subpackage and must be copied to the user-specified volume and script writer-specified directory before passing control to Batch to execute the installation script. These file names must also be in the *:ProductFiles:* or *:PublicProductFiles:* list. The Installation Manager performs the copy operation. Several files cannot be included in the *:PreInstallFiles:* list. New versions of the *InstallMgr.run*, *Batch.run*, and *BatchMsg.bin* must be copied before Installation Manager is invoked.

Installation Script File

The *installation script file* contains Batch JCL instructions that facilitate the installation of a subpackage. As you write this JCL file, you should be aware that most installation errors occur when Batch executes the instructions in this file.

Since you should allow the user to install your subpackage publicly, the script file(s) must handle installation from a server, as well as installation from the distribution media (floppy or tape).

Your installation script file can use multiple JCL files by using the $Call command. While this practice can make your installation scripts more modular, ease script development, and speed up the user's software installation process, it is not recommended. The subpackage concept, the use of the installation variables, and the use of a message file reduce the amount of work that a JCL file has to do and the need for overly large JCL scripts. If you do use multiple JCL files, you are responsible for making them accessible to Batch for all types of installations, including installations from the server. Also, the additional JCL files will not be deleted when a package is removed. The main JCL file, message file, and command file are deleted because the Installation Manager knows their names.

Note: *Executive video commands, such as Video and Screen Setup, should not be used in installation script files. Using the Batch commands to control video adversely affects the screen that is set up by Installation Manager.*

Message File

Other than two exceptions, all displayed nationalizable text strings belong in a *message file*. This aids nationalization efforts. After creating the message file, use the Executive command, **Create Message File**, to produce the binary message file for the distribution media.

Two nationalizable string exceptions that belong in your script file. The first is used to report the absence of the message file, and the second is used as an argument to NextFloppy() to ensure that the correct disk is mounted.

Command File

The *command file* for a software product contains all the commands that need to be added to the system in support of that product. Each product command file is a standard command file. such as the Executive command file, *Sys.cmds*, but usually with fewer entries in it. Don't forget to place all the command names in the *Commands* field of the control file.

Use the Executive command **Command File Editor** to create the file containing the commands that are used to invoke your run file(s) and/or install your system service. You should use the **Merge Command Files** command statement in your script file to update the user's command file. We recommend that you do not use **New Command** in the script file.

Note: *The command file is required only if your product has associated commands.*

The command file for Mouse Services, for example, contains the following entries (commands):

Install Mouse Service
Deinstall Mouse Service
Set Mouse Controls

For more details on how to create a customized command file, see the description of the **Command File Editor**, in the *CTOS Executive Reference Manual.*

The Installation Manager does not read the contents of a command file. It merely copies the file to a temporary location so that the file contents are available to Batch. During the installation process, your script file, using the installation variables *CmdFileFrom*, *CmdFileTo*, and *VolumeTo*, tells the **Command File Editor** to merge the commands in this file with the user-specified command file.

Because a command file merge adds all the commands to the user-specified command file, you have the option of creating several command files for your product. By doing so, each command file you create can be merged with the command file of the user. This allows you to selectively merge only those commands the user may need. However, if you use more than one command file, your script is responsible for placing it somewhere where it will be available to all types of installations, including installation from server.

For public installations, your script should also merge the command file into *[!sys]<sys>sys.cmds*. Otherwise, the application can not be executed at the server.

User Configuration File

In addition to reading the installation files just described, the Installation Manager reads the contents of the user configuration (*.user*) file. It then sets the following values of the installation variables:

- Installation type (public or private)
- Video level (verbose or silent)
- Whether to archive the currently installed version of the subpackage
- Whether to save the archive
- The volume and directory into which to create the archive (if archiving is desired) when the installation is private
- The volume and directory into which to create the archive (if archiving is desired) when the installation is public
- The volume on which to install the software when the installation is private
- The volume on which to install the software when the installation is public
- The name of the Context Manager configuration file to use when the installation is private
- The name of the Context Manager configuration file to use when the installation is public
- Whether to use a log file
- The name of the log file

The variable name associated with each of these functions and its description are included later in this section.

As a script writer, you do not have to be concerned about how the user configuration file (or the user interactively) influences an installation. There is one parameter, however, that you need to take into account: whether the installation is public or private. You only need to be aware of this value if the distribution media is floppy disks and the files for a public installation are not the same as for a private installation. In all other cases, by using the installation variables provided by the Installation Manager to Batch, this attribute is transparent to you.

Naming Your Floppy Installation Files

In the simplest case, the product is atomic. It has one name in the database. In this case, the names of the installation files are

Install.ctrl
Install.jcl
InstallMsg.bin
Install.cmds

With subpackages, you have at least two levels of control files. The top level is always called *Install.ctrl*. To illustrate, let's use the example of an accounting package, Accounts for You, pictured in Figure 6-1.

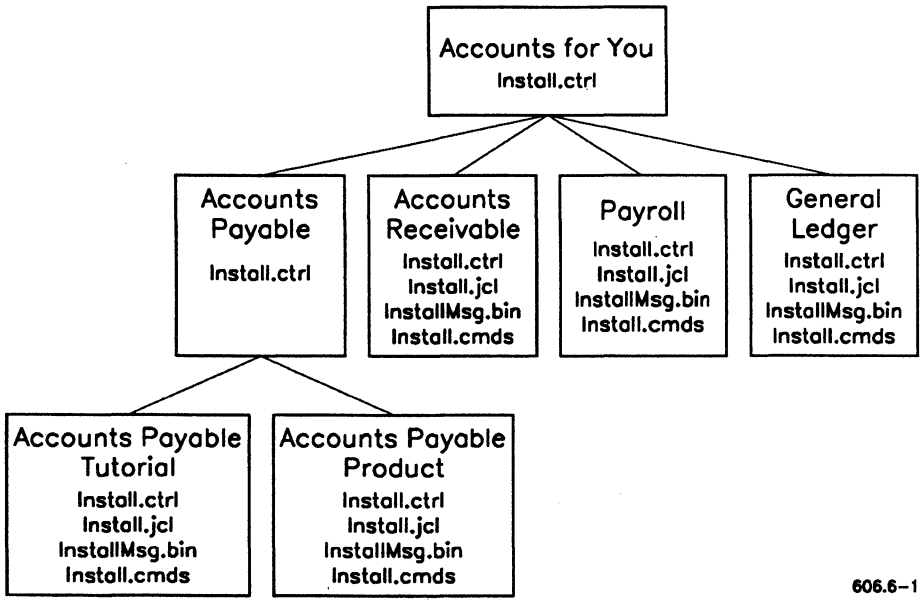
The *Install.ctrl* file for Accounts for You would have the following entries:

:PackageName: Accounts for You
:MultiPkgName: Accounts Payable
:MultiPkgName: Accounts Receivable
:MultiPkgName: Payroll
:MultiPkgName: General Ledger

The Accounts Receivable subpackage has its own control file, installation script, message file, and command files, as shown in the figure. So do the Payroll and General Ledger subpackages. They are called

AccountsReceivable>Install.ctrl
AccountsReceivable>Install.jcl
AccountsReceivable>InstallMsg.bin
AccountsReceivable>Install.cmds
Payroll>Install.ctrl
Payroll>Install.jcl
Payroll>InstallMsg.bin
Payroll>Install.cmds
GeneralLedger>Install.ctrl
GeneralLedger>Install.jcl
GeneralLedger>InstallMsg.bin
GeneralLedger>Install.cmds

Figure 6-1. Files for Floppy Installation of Accounting Package



The Accounts Payable subpackage is further subdivided into two subpackages: Accounts Payable Tutorial and Accounts Payable Product. Accounts Payable has only a control file, since it is not yet at the bottom of the product "tree." Its two subpackages each have all four installation files, since they are at the bottom of the tree. The *AccountsPayable>Install.ctrl* file would have the following entries:

:PackageName: Accounts Payable
:MultiPkgName: Accounts Payable Tutorial
:MultiPkgName: Accounts Payable Product

In this example, there are two layers before the user actually chooses a subpackage. The user is presented a menu containing Accounts Payable, Accounts Receivable, Payroll, and General Ledger. After Accounts Payable is chosen, another menu with the contents of *AccountsPayable>Install.ctrl* is displayed, and the user chooses which of those subpackages to install.

During installation, the Installation Manager reads *Install.ctrl* and displays a menu. When the user makes a selection (let's say that all five items are chosen), the Installation Manager tries to open the control file related to the first (top) entry, *AccountsPayableTutorial>Install.ctrl*. If this file is not found, it prompts the user to mount the proper disk. If it is found, the installation files (*AccountsPayableTutorial>Install**) are copied to *[Scr]<$>*. The control file is read and information is extracted. Note that the *:PackageName:* entry must match the prefix part of the name of the control file. Control is then passed to Batch, which executes *AccountsPayableTutorial>Install.jcl*. When the first subpackage has been successfully installed, the Installation Manager then tries to read *AccountsPayableProduct>Install.ctrl*, and the process repeats itself.

Note: *The Installation Manager makes available all the above files (except the control file), but no others, to Batch, by copying them to* [Scr]<$>. *If the installation is public, the Installation Manager also copies them to* [!Sys]<Installed>PkgName>Version>Install*, *which makes them available when a user installs the package from the server. If your script calls other scripts, uses other command files, or uses any other files, it is your responsibility to make the script copy them from the floppy disk to where they are accessible for later use.*

Naming Your Tape Installation Files

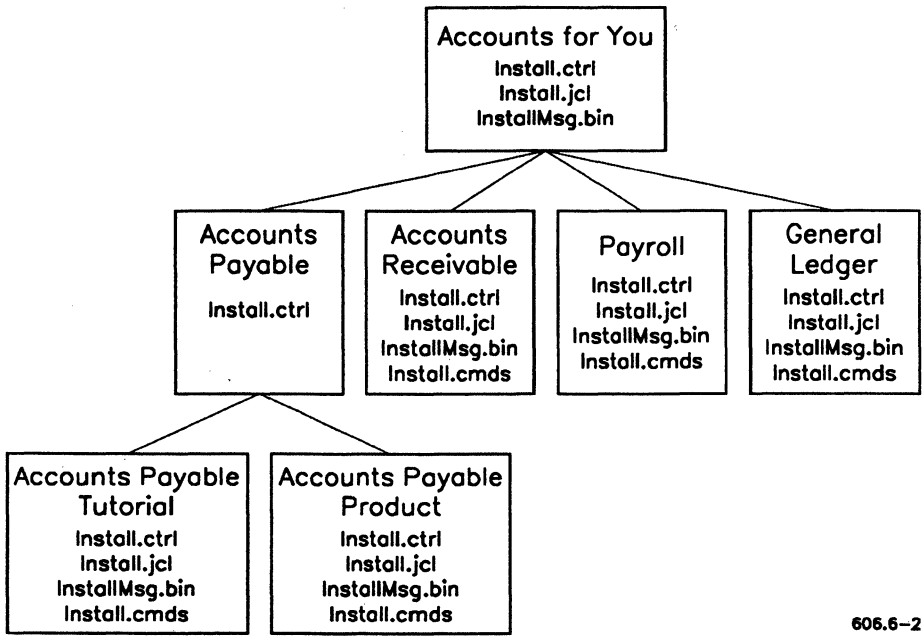
The method of installing a product from tape is slightly different from a floppy disk installation and therefore requires a slightly different scheme.

Tape installation requires all of the control files and command files described earlier in this section. However, it uses only one message file and one script file, *InstallMsg.bin* and *Install.jcl*. Figure 6-2 shows the installation files required for a tape installation for the product described in Figure 6-1. All twenty-four installation files should be archived to the first sequential access device, such as *[Qic]0*.

Note that *Install.jcl* must install all the subpackages that the user selects. It needs to query the installation variable, *Pkgs*, to determine the packages that the user has chosen. The individual installation control files are used just as they are in a floppy installation. The individual installation script, message, and command files (for example, *Payroll>Install.jcl*, *Payroll>InstallMsg.bin*, and *Payroll>Install.cmds*) are not used by the tape installation but are included in case the user chooses to do a public installation. In this case, the script, message, and command files are copied to *[!Sys]<Installed>* for subsequent use when someone does an installation from server.

Note: *The Installation Manager places all the files in the installation archive file* ([Qic]0) *into* [Scr]<$>. *If your script uses other files, it is your responsibility to have the script move them to permanent location so that they are accessible when the user installs your product from the server.*

Figure 6-2. Files Used in a Tape Installation for an Accounting Package



Organizing Your Installation Media

Once you have created the files you need for your program and its installation, you place these files on the appropriate distribution media.

For a Floppy Installation

For floppy disks, the control file, installation script, message file, and command file must reside in the <*Sys*> directory of the floppy disk. If your product contains only one subpackage, the installation files go onto the first disk. However, if you have more than one subpackage, the *Install.ctrl* file goes on the first disk, and the installation files go on the first disk that contains the product files associated with that subpackage.

Each disk other than the first requires an empty file whose name identifies the disk. You must place the file names in InstallMsg.bin. The disk name is used to request the next disk and verify that the correct disk has been inserted into the drive.

The following list shows the installation files required for a product that contains three subpackages: a word processor, a spell checker, and a grammar checker.

Disk Number	Subpackage Name	Installation Files
1	Word Processor	*Word Processor Disk 1 of 7* *Install.ctrl* *WordProcessor>Install.ctrl* *WordProcessor>Install.jcl* *WordProcessor>Install.cmds* *WordProcessor>InstallMsg.bin* *SpellChecker>Install.ctrl* *SpellChecker>Install.jcl* *SpellChecker>Install.cmds* *SpellChecker>InstallMsg.bin* *GrammarChecker>Install.ctrl* *GrammarChecker>Install.jcl* *GrammarChecker>Install.cmds* *GrammarChecker>InstallMsg.bin*
2	Word Processor	*Word Processor Disk 2 of 7*
3	Word Processor	*Word Processor Disk 3 of 7*
4	Word Processor and Spell Checker	*Word Processor Disk 4 of 7* *SpellChecker>Install.ctrl* *SpellChecker>Install.jcl* *SpellChecker>Install.cmds* *SpellChecker>InstallMsg.bin*
5	Spell Checker	*Word Processor Disk 5 of 7*
6	Spell Checker	*Word Processor Disk 6 of 7*
7	Grammar Checker	*Word Processor Disk 1 of 7* *GrammarChecker>Install.ctrl* *GrammarChecker>Install.jcl* *GrammarChecker>Install.cmds* *GrammarChecker>InstallMsg.bin*

When two subpackages fit on one disk, all the installation files for those subpackages must be placed in *[sys]* on that disk.

It is convenient for the user if all installation files are duplicated on the first disk, although this is not required. If this is done, the user has a better idea of what disk to insert. For example, suppose that the user inserts the first disk and invokes the Installation Manager. A menu is then presented allowing him or her to choose the Word Processor, Spell Checker, or Grammar Checker, and the Grammar Checker, is chosen. The Installation Manager tries to open *GrammarChecker>Install.ctrl*. Since it is not on the first disk, the Installation Manager prompts the user with a message such as, "Please insert the first disk of the Grammar Checker software." The user must then look in the Software Release Announcement to find out which of the seven disks is the first disk of the Grammar Checker.

However, if all the installation files are placed on Disk 1 and on the other disks as described above, *GrammarChecker>Install.ctrl* is found, the files are copied to *[Scr]<$>*, and Batch is invoked. The first statement in the installation script should always be a NextFloppy statement for the proper disk, in this case: NextFloppy('Word Processor Disk 7 of 7'). The user is prompted for the proper disk.

Note that even when all the installation files are placed on the first disk, the installation files for the Spell Checker and the Grammar Checker must still go on Disk 4 and Disk 7 respectively. The reason is that the user may choose more than one subpackage, and only one subpackage installation file is copied to *[Scr]<$>* at a time. If the user chooses all three subpackages, *WordProcessor>Install** is copied to *[Scr]<$>*, control is passed to Batch, and the Installation Manager updates the database, all before the Installation Manager tries to open *SpellChecker>Install.ctrl*. Since the user has disk 4 mounted, it is logical to put it there instead of making the user reinsert disk 1 just to get the installation files.

For a Tape Installation

For tape the installation files should be placed in *[Qic]0*. By placing all your product files on the next tape file, (in this case, *[Qic]1*), you are able to take advantage of the *DeviceFrom* variable in your script file. In general, even numbered tape files should contain installation files and the next tape file should contain product files. However, as we will show later, even the most complicated tape installation can be easily done by using only *[Qic]0* and *[Qic]1*.

After the user has had the opportunity to change the installation variable values, the Installation Manager uses *RestoreArchive.run* to place all the files from the installation archive file (usually *[Qic]0*) into *[Scr]<$>*. The Installation Manager reads the control files as necessitated by the user's choices. It builds lists of files that belong to the chosen subpackages and puts them in files in *[Scr]<$>* where they are available to the script. It then passes control to Batch, setting *Install.jcl* as the script file and *InstallMsg.bin* as the message file. The script file performs the actual installation. When the Installation Manager is reloaded, it checks that all subpackages were installed correctly and updates the database for each subpackage. If the verify feature is enable, it checks that the product files time stamp is later than the time that Installation Manager was first invoked, and also checks the the files in the *OverwriteNoFiles* list exist.

Installation Variables

The *CTOS Batch Manager II Installation and Configuration Guide* contains most of the following information. It is repeated here with additional information relating specifically to the installation process. These installation variables are set by the user through the Installation Manager before the installation script is processed by Batch. They should not be changed by the installation script. Since the values are passed to Batch by the Installation Manager, the values are only valid when Batch has been invoked by the Installation Manager. Since all these variables can be set by the user, you must use them if they apply to your installation. For example, you cannot assume that if an installation is public, that your files are being copied to *[!Sys]*; You must use the variable *VolumeTo*. Likewise, you cannot assume that for private installations your commands should be merged into *[Sys]<Sys>Sys.cmds*. You must use *CmdFileTo*.

These variables are read-only. Although Batch does not report an error if your script assigns a value to them (for example, VolumeTo = *'[Sys]'*), this approach may thwart the desires of the user and is not recommended.

CmConfigFile

Is the name of the user's Context Manager configuration file. The name is set by the user during the examine/change defaults phase of the installation. It should be used as the argument to the *CM fonfig file* parameter in **CM Add Application**. The default is *[Sys]<Sys>CmConfig.sys*.

CmdFileFrom

Is the name of the command file that contains the commands for the subpackage. It is set to Subpackage>*Install.cmds*. It should be used as the argument to the *Command file from* parameter in the **Merge Command Files** command.

CmdFileTo

Is the name of the command file into which the commands in *CmdFileFrom* will be merged. The name is set by the user during the examine/change defaults phase of the installation. However, if the installation is public, the value is always *[!Sys]<Sys>Cluster.cmds*. It should be used as the argument to the *Command File To* parameter in the **Merge Command Files** command.

DeviceFrom

Is the floppy disk drive in which the installation disk is mounted (5.25 inch and 3.5 inch drives). The value is of the form *[fn]* where n is the drive number. For tape installations, this variable is the tape specification. The value is one greater than the tape specification given by the user. In other words, if the user did not change the tape specification from its default of *[Qic]0*, its value would be *[Qic]1*, the tape file where your product files are archived. It should be used as the argument to the *Archive dataset* parameter of **Restore Archive**. For server installations, it is the node name (*{Master}*) and volume name of the volume to which it was publicly installed. (The Installation Manager keeps track of it in the database).

DirectoryTo

Is the directory name, including angle brackets, into which the product files will be copied. If your product files must be copied into more than one directory, you should disregard this variable. It is always <*Sys*> for tape installations. This is not used with version 12.0.

FloppyType

Is the type of floppy media if disks are used. A value of 1 indicates 5.25 inch disks and 4 indicates 3.5 inch disks.

ImVersion

Is the Installation Manager enhancement version. ImVersion is 1 in Installation Manager/Batch Version 12.3 and will be incremented every time a new script feature is made available (not necessarily with each release). You can use this variable when the lowest supported version of Installation Manager/Batch is 12.3. By querying this variable with version 12.3 or greater, you can tell what features are available and avoid syntax errors caused by references to variables not supported in earlier versions of Installation Manager/Batch.

InstallType

Is the type of installation that the user is performing:

0 = Floppy
1 = Tape
2 = Server

MsgFile

Is the name of the message file associated with the subpackage. It is meant to be used as an argument to the Batch statement InitMsgFile. For tape installations it is always *InstallMsg.bin*. This is not used with version 12.0.

Pkgs

Is a string containing the subpackage currently being installed and all subsequent subpackages that the user has chosen. Calls to the Batch statement SubString should be used to query the contents of this variable. If other subpackages have been chosen, you can prompt the user to insert the correct disk to continue the installation.

Public

Is a flag set to TRUE if the user has chosen a public installation. Otherwise, it is set to FALSE. Note that the proper way to interrogate a flag in an If statement is:

```
If Public
...
EndIf
```

Do not use:

```
True = 65535
If Public = True
...
Endif
```

because you may set the value of TRUE incorrectly, and it takes up space in Batch's symbol table.

Unattended

Is a flag used to indicate that the user does not want to be queried during the installation script. This variable should always be queried before performing the Batch functions DisplayAndWait, UserEnterValue, UserSelectMultiple, UserSelectSingle, and UserSelectYesNo. It is always FALSE during a floppy disk installation.

VolumeTo

Is the volume name, including square brackets, of the volume onto which the user wants the subpackage(s) to be installed. It must also be used as the argument to the *Default volume* parameter in the **Merge Command Files** command if subsequent installations from a server are to work properly. This is not used with version 12.0.

File Lists

The Installation Manager also creates four at-files that are extremely useful for tape and server installation, as described below.

[Scr]<$>InstallOverwriteNoFrom.fls

This is a list of file that are listed in the control file entry *:OverwriteNoFiles:* (or *:PublicOverwriteNoFiles:* if the installation is public).

For floppy and server installations, this list contains only files in the subpackage that are currently being installed. The file specifications contain volume, directory, and file names. For floppy installations, the volume is the same as *DeviceFrom* and the directory is the same as *DirectoryTo*. Since *DirectoryTo* is usually *<Sys>*, and the product files are usually (and should be) placed in a directory other than *<Sys>*, the list is not very useful for floppy installations. For server installations, the values for volume and directory are taken from the server's installation database.

For tape installations this list contains files in every subpackage selected by the user. The file specifications are of the form *<*>FileName*, making the list suitable as an argument to the *[File list from (<*>*)]* parameter of the Executive command **Restore Archive**.

[Scr]<$>InstallOverwriteNoTo.fls

This is a list of files that are listed in the control file entry *:OverwriteNoFiles:* (or *:PublicOverwriteNoFiles:* if the installation is public). The list contains the same files as *InstallOverwriteNoFrom.fls*, but the volume and directory information are different.

For floppy and server installations, this list contains only files in the subpackage that is currently being installed. The file specifications contain volume, directory, and file names. The volume is the same as *VolumeTo* and the directory is the same as *DirectoryTo*.

For tape installations this list contains files in every subpackage selected by the user. The file specifications contain volume, directory, and file names. The volume is the same as *VolumeTo* and the directory is the same as *DirectoryTo*, making the list suitable as an argument to the *[File list to (<*>*)]* parameter of the Executive command **Restore Archive**.

[Scr]<$>InstallOverwriteOkFrom.fls

This is a list of files that are listed in the control file entry *:ProductFiles:* (or *:PublicProductFiles:* if the installation is public). See the description for *InstallOverwriteNoFrom.fls*, above.

[Scr]<$>InstallOverwriteOkTo.fls

This is a list of files that are listed in the control file entry ProductFiles (or PublicProductFiles if the installation is public). See the description for *InstallOverwriteNoTo.fls*, above.

Restarting an Installation

The sample installation scripts at the end of this section contain a number of restart labels ("RestartLabel"). Restart labels are markers within the JCL file that indicate points where processing of the JCL file can be restarted. This subsection includes a brief description of how Batch and the Installation Manager work with restart labels.

Every time Batch encounters a restart label, it opens the JCL file header and writes the number of restart label statements it has encountered. After each nonintrinsic statement Batch encounters (such as a Command statement or a Run statement), it looks in the ASCB to see if the application completed successfully. If it did not complete successfully (and *ContinueOnError* was not specified), Batch exits and loads the Installation Manager. The Installation Manager looks in the ASCB for a nonzero status code. It then opens the JCL file that was being processed and looks at the file header to see if it can be restarted (that is, the number of restart labels is greater than zero). The Installation Manager puts some additional information into the file header, reports the error to the user, and exits.

The user looks in the log file and fixes the problem. For example, if the directory was full, creating a bigger directory or deleting some files might help.

The user then invokes the Installation Manager. The Installation Manager looks for JCL files, opens the file and checks the file header to see whether it can be restarted. If it can be restarted, the Installation Manager tells the user that the last installation failed and allows her or him to restart it.

If the user chooses to restart the previous installation, control passes to Batch, which opens the file and looks for the last restart label it encountered. It then begins processing the JCL file from that point.

Note that the presence of restart labels in a JCL file does not guarantee that the installation can be restarted in all cases. First, the error must have occurred after Batch has read at least one RestartLabel statement in the JCL file. Second, the utility reporting the error must place the status code in the ASCB. (Otherwise the Installation Manager has no way of knowing that an error occurred.) Third, Batch must have time to write the necessary information in the file header of the JCL file before it returns control to the Installation Manager.

Nationalization

The file *[Sys]<Sys>Install>English.cmds* contains common commands (in English) used in installation scripts. This file is provided so that the Command statements used in installation scripts do not have to be translated from English into the local language.

Check to be sure that all the command names that you use with Command statements are contained in *[Sys]<Sys>Install>English.cmds*. If a command used in your script is not in this file, add the command through use of the **Merge Command Files** command or the **New Command** command (in your script file). Do not overwrite the existing *Install>English.cmds* file, because other products might rely on commands they have added to it.

The commands that are in the 12.3 version of *Install>English.cmds* are **Append, Batch Foreground, CM Add Application, CM Remove Application, Configure Disk, Copy, Create Directory, Create File, Create Message File, Delete, Expand File, Floppy Install, Format Disk, Install New Requests, Install Sequential Access Service, Install Xbif Service, Installation Manager, LCopy, List Request Set, Logout, Make Request Set, Merge Binary Message Files, Merge Command Files, Move, New Command, Path, Remove Command, Remove Directory, Rename, Restore Archive, Run, Selective Archive, Set Protection, Submit, Tape Install, Type, Update Request Set,** and **Volume Archive**.

Installation Tips

The following paragraphs present some tips for creating installation scripts.

Installation Manager Version 12.0 and Earlier

- If your product has to be installable using version 12.0 of the Installation Manager, you cannot use subpackages. To make your installation script compatible with both versions, your installation script must perform the extra work of assigning values to those installation variables that are not available in version 12.0. Use the Batch FileVersion statement to determine what version of the Installation Manager run file you have. The variables that you have to set are *DirectoryTo*, *MsgFile*, and *VolumeTo*.

  FileVersion returns a string. It should be checked as follows:

```
ver = FileVersion('[Sys]<Sys>InstallMgr.run')
If Substring('12.0',ver)<>65535
  ...
EndIf
```

  By calling Substring instead of checking to see if ver = '12.0,' you are assured of obtaining the correct answer regardless of the version of 12.0 that the user may have installed. (Substring would report TRUE when the version is 12.0.1 where as ver = '12.0' would not.)

- You may need to write an installation script that is compatible with Installation Manager Version 12.0. To accomplish this, follow these steps:

  1. Create *install.ctrl* and *install.jcl* installation files for version 12.0 Batch/Installation Manager.
  2. Create installation files for version 12.1 or greater Batch/ Installation Manager only. Use subpackages even if you only have one package. Name your supporting files Subpackage>*Install.ctrl*, etc. See "Naming Your Floppy Installation Files" above.
  3. Place one *:MultiPkgName:* token per subpackage in the version 12.1 or greater *Install.ctrl*.

During execution, the 12.0 Installation Manager reads *Install.ctrl*, ignores the *:MultiPkgName:* token(s), and passes *Install.jcl* to 12.0 Batch. If 12.1 or greater Installation Manager is used, it reads only the *:MultiPkgName:* token(s) from Install.ctrl, ignores the information for the 12.0 Installation Manager, and presents the user with a menu. The Installation Manager then passes the selected subpackages' *Install.jcl* scripts to 12.1 Batch.

If the product contains only one subpackage, the user is presented with a menu with only one item in it. While this may seem a bit awkward, it does make the debugging of the scripts easier by isolating the files used by version 12.0 and versions 12.1 and later.

Using Variables

- Do not use hardcoded values in installation scripts. Use the variables and tokens provided by Installation Manager whenever possible.
- Do not overwrite the Installion Manager variables with your own data. If you need different values, use your own variables. Make use of the values set during the installation configuration whenever possible.
- Do not hardcode drives or volumes in your installation script. Use the variables *DeviceFrom* and *DeviceTo*. This allows installation from any drive and to any drive or volume.
- Merge the command file specified by the user by using the installation variables *CmdFileFrom* and *CmdFileTo*. The exception to this is a public installation. Merge the commands to *[!Sys]<Sys>Sys.cmds* to make them accessable by the client workstation and at the server.

Messages

- Use message files rather than placing messages directly into the installation script. This aids nationalization. The only exceptions are reporting the absence of a message file and ensuring that the correct disk is mounted.
- Avoid asking questions. When it is necessary to get information from the user, ask the questions at the beginning of the script. Once installation begins, the only interaction required by the user should be for media changes.
- Display a message for each action. For example, when the user presses a key to continue, acknowledge the action with a message.
- Do not ask the user for information that Installation Manager has already obtained (the name of the command file or Context Manager configuration file, for example).
- Avoid asking yes/no questions. Make use of menus instead.

Packages and Subpackages

- If you are using multiple packages, use the Batch variable *pkgs*. This allows the installation process to skip packages that were not selected and take the user directly to the next required package to be installed.
- Use subpackages if there are any parts of a package that can be installed separately.

Volumes and Directories

- Do not change the path anywhere within the installation script. Because you have no way of knowing the user's current working directory, there is no way to return. Use the installation variables to create destinations instead.
- Installation can start from any disk as long as that disk has the required Installation Manager files.
- If you need to install to multiple directories that cannot be split into subpackages, use a volume/directory specification with each product file specified instead of using the *DirectoryTo* variable. This is the only case where you should use volume directory specifications instead of *DirectoryTo*.
- Do not use **Volume Status** to find the space available on the disk. Use RequiredDiskSpace in the control file

File Lists

- Use the *:Verify:* token and the *:ProductFiles:* token to verify successful installation be checking file dates.
- Do not create or supply your own temporary file list. Use the file lists in the control file and then use the associated .fls created by Installation Manager.
- When using a list of of files, separate each file with a space. Do not use a carriage return. File names can include spaces if the name is enclosed in single quotes.

Other Tips

- If you give the user the option to terminate during the script, use the Batch Cancel statement. When an installation terminates abnormally, Batch needs to notify the Installation Manager not to update the database. If you do not use the Batch command Cancel statement, Batch will have no way of knowing that the script was terminated abnormally and the database could be incorrectly updated. When Batch encounters a Cancel statement, it causes the Installation Manager to report status code 4 ("Operator intervention").
- Avoid using nested job control language (jcl) files, which in turn allows you to avoid the necessity of using the Call command. Using the Call command can cause Batch to terminate with status code 302 ("Write protected") if it encounters a RestartLabel on write protected media. If your product requires multiple components, break them into separate subpackages instead. You should use the Call command to install multiple components only if your Installation Manager and Batch are version 12.0 or earlier. Installation Manager 12.0 does not support subpackages.
- If, during testing, your script fails with "Command not found," you are using a command that is not contained in *Install>English.cmds*. In that case your script must add the command using the **New Command** command before it issues the Command statement for that command.

Installation Script Examples

The following pages illustrate three sets of installation files. Each JCL installation script file begins on a lefthand page. Numbered comments accompanying each script correspond to script lines. The facing righthand page includes comments for the numbered lines in the script. These examples are not intended to teach you how to write JCL, but instead, how to use the installation variables to best advantage.

The binary message file is created from a text file. The contents of the text file is shown, even though the file is labeled **InstallMsg.bin*.

The values of the installation variables are given as if the user had not changed them from their default values. For example, *VolumeTo* will be described as having the value *[Sys]* for private installations. As stated above, you as script writer cannot count on any specific value being in the installation variables, but instead must use them to carry out the desires of the user.

Example 1: One Subpackage

This example contains one subpackage and spans two floppy disks. The installation files are *Install.ctrl*, *InstallMsg.bin*, *Install.cmds*, and *Install.jcl*. They are placed in the <*Sys*> directory of the first disk. A zero-length file, 'Tape Utilities Diskette 1 of 2' is put in the <*Sys*> directory on the first disk. A zero-length file, 'Tape Utilities Diskette 2 of 2' is put in the <*Sys*> directory on the second disk. Product files are placed in the <*CTOS*> directory of the two disks.

Install.ctrl File

```
:PackageName:Basic Tape
:Version:12.3.0
:RequiredDiskSpace:674
:Verify:
:Directory:<Sys>
:Commands:'Configure Sequential Access Device' 'DeInstall
Sequential Access Service' 'Install Sequential Access
Service' 'Tape Copy' 'Tape Erase' 'TapeRetension'
:ProductFiles:InstallSeqService.run NGenSeqService.run
QicConfig.sys SeqAccessCopy.run SeqAccessUtility.run
SeqServiceMsg.bin
```

(*QicConfig.sys* is not placed in OverwriteNoFiles because in this release, the format of the file has changed. Files in the old format will no longer work, and therefore must be overwritten.)

InstallMsg.bin File

```
:1: "Tape Utilities Diskette 2 of 2"
:2: "Merging commands... "
:3: "Copying files... "
:4: "done."
:5: "yes"
:6: "no"
```

Install.cmds File

The **Command File Editor** was used to create the six commands listed in the control file.

Install.jcl File

Number	Script
	`; Install.jcl`
	`; Install the Basic Tape subpackage`
	`; Except for yes/no strings, this message is the`
	`; only nationalizable string in this file.`
1	`ErrorMsg = 'Cannot access message file'`
2	`FromFloppy = 0`
	`FromServer = 2`
3	`FloppySpec = ConcatStrings(DeviceFrom,'<CTOS>')`
	`FromSpec = ConcatStrings(FloppySpec, '*')`
	`ToSpec = ConcatStrings(VolumeTo, DirectoryTo)`
4	`InitErc = InitMsgFile(MsgFile)`
	`If InitErc > 0`
	`  DisplayLine(ErrorMsg)`
	`  Cancel`
	`Endif`
5	`CopyMsg = GetMsg(3)`
	`DoneMsg = GetMsg(4)`
	`Msg = GetMsg(2)`
	`Display(Msg)`
6	`Command Merge Command Files ,&`
	`  CmdFileFrom ,&`
	`  CmdFileTo ,&`
	`  Yes ,&`
	`  VolumeTo`
	`DisplayLine(DoneMsg)`
	`DisplayLine`
	`Display(CopyMsg)`

Number	Comment
	In this example, when values for installation variables are given, it is assumed that the user has not used the *Examine / Change Defaults* option from the Installation Defaults menu. For floppy disk installations, it is assumed that the user has inserted the floppy disk in drive *[f0]*.
1.	Assign strings to be used later.
2.	Assign values to enhance readability.
3.	DeviceFrom is *[f0]* if the installation is private, *[!Sys]* if the installation is public. VolumeTo is *[Sys]* if the installation is private, *[!Sys]* if public. DirectoryTo is *<Sys>*.
4.	Initialize the message file. The message file is *[Scr]<$>InstallMsg.bin* if the installation is private or *[!Sys]<Installed>BasicTape>12.3.0>InstallMsg.bin* if the installation is from the server. Print a message if the file does not exist. It should always be there; this is just defensive coding. The Cancel statement causes termination of script processing by Batch. The installation will fail because the product files were not updated.
5.	Since the "copying" and "done" messages are used more than once, put them in their own variables.
6.	Merge the command files. *CmdFileFrom* is *[Scr]<$>Install.cmds* if the installation is private or *[!Sys]<Installed>BasicTape>12.3.0>Install.cmds* if the installation is from the server. *CmdFileTo* is *[Sys]<Sys>Sys.cmds* if the installation is private, *[!Sys]<Sys>Cluster.cmds* if it is public. *VolumeTo* is *[Sys]* if the installation is private, *[!Sys]* if it is public.

Number	Script
7	If InstallType = FromServer
	Command LCopy ,&
	,&
	@[Scr]<$>InstallOverwriteOkFrom.fls,&
	@[Scr]<$>InstallOverwriteOkTo.fls,&
	,&
	Yes ,&
	No ,&
	No ,&
	Yes
	DisplayLine(DoneMsg)
	DisplayLine
8	GoTo Endit
	Endif
	RestartLabel
9	Command LCopy ,&
	FromSpec ,&
	,&
	ToSpec ,&
	,&
	Yes ,&
	No ,&
	No ,&
	Yes
	DisplayLine(DoneMsg)
	DisplayLine
	RestartLabel

Number	Comment
7.	When the user does a public installation, this script file, along with the accompanying command and message files are copied to *[!Sys]<Installed>* with the prefix *BasicTape>12.3.0>*. The script is invoked when the user attempts to install the product from the server. Therefore it (and indeed every script) must allow for the case when the user installs from the server (even if the script displays a message saying that installation from the server is not supported for the subpackage). This **LCopy** command is performed if the user invoked the Installation Manager using the Executive command **Server Install** or if the *Install from server* option was chosen from the Installation Media menu. *[Scr]<$>InstallOverwriteOkFrom.fls* contains: *[!Sys]<Sys>InstallSeqService.run* *[!Sys]<Sys>NGenSeqService.run* *[!Sys]<Sys>QicConfig.sys* *[!Sys]<Sys>SeqAccessCopy.run* *[!Sys]<Sys>SeqAccessUtility.run* *[!Sys]<Sys>SeqServiceMsg.bin* *[Scr]<$>InstallOverwriteOkTo.fls* contains: *[Sys]<Sys>InstallSeqService.run* *[Sys]<Sys>NGenSeqService.run* *[Sys]<Sys>QicConfig.sys* *[Sys]<Sys>SeqAccessCopy.run* *[Sys]<Sys>SeqAccessUtility.run* *[Sys]<Sys>SeqServiceMsg.bin*
8.	Since there is nothing more to do in the case of an installation from the server, quit. A GoTo statement is used to make the script more readable. An Else clause would have been equally correct.
9.	Everything from this point gets executed only if the installation is being done from floppy disks. Copy the files from the first disk. *FromSpec* is *[f0]<CTOS>** and *ToSpec* is *[Sys]<Sys>* if the installation is private or *[!Sys]<Sys>* if the installation is public.

Number	Script

```
10      Msg = GetMsg(1)
        NextFloppy(Msg)

        DisplayLine
        Display(CopyMsg)

11      Command LCopy                   ,&
            FromSpec                    ,&
                                        ,&
            ToSpec                      ,&
                                        ,&
                Yes                     ,&
                No                      ,&
                No                      ,&
                Yes

        DisplayLine(DoneMsg)
        DisplayLine

        Endit:
        End
```

Number	Comment
10.	Ask the user to insert the next floppy disk.
11.	Copy the files from the second disk. *FromSpec* and *ToSpec* remain the same.

Example 2: Nested Subpackages

This second example contains two levels of subpackages and spans three floppy disks. The installation files are

Install.ctrl
DevelopmentUtilities>Install.ctrl

DevelopmentRunFiles>Install.ctrl
DevelopmentRunFiles>Install.jcl
DevelopmentRunFiles>InstallMsg.bin
DevelopmentRunFiles>Install.cmds

DevelopmentLibraries>Install.ctrl
DevelopmentLibraries>Install.jcl
DevelopmentLibraries>InstallMsg.bin

AsynchronousExamples>Install.ctrl
AsynchronousExamples>Install.jcl
AsynchronousExamples>InstallMsg.bin
AsynchronousExamples>Install.cmds

These files are placed in the <*Sys*> directory of the first disk.

The following files are also placed in the <*Sys*> directory of the Disk 2:

DevelopmentLibraries>Install.ctrl
DevelopmentLibraries>Install.jcl
DevelopmentLibraries>InstallMsg.bin

The following files are also placed in the <*Sys*> directory of the Disk 3:

AsynchronousExamples>Install.ctrl
AsynchronousExamples>Install.jcl
AsynchronousExamples>InstallMsg.bin
AsynchronousExamples>Install.cmds

A zero-length file, 'Development Package Disk 1 of 3' is put in the <*Sys*> directory on the first disk. A zero-length file, 'Development Package Disk 2 of 3' is put in the <*Sys*> directory on the second disk. A zero-length file, 'Development Package Disk 3 of 3' is put in the <*Sys*> directory on the third disk.

Product files for Development Run Files are placed in the <*CTOS*> directory of Disks 1 and 2. Product files for Development Libraries are placed in the <*Lib*> directory of Disks 2 and 3. Product files for Asynchronous examples are placed in the <*Async*> directory of Disk 3.

Install.ctrl File

```
:PackageName:Development Package
:MultiPkgName:Development Utilities
:MultiPkgName:Asynchronous Examples
```

DevelopmentUtilities>Install.ctrl File

```
:PackageName:Development Utilities
:MultiPkgName:Development Run Files
:MultiPkgName:Development Libraries
```

Note that Install.ctrl could have been designed to look like:

```
:PackageName:Development Package
:MultiPkgName:Development Run Files
:MultiPkgName:Development Libraries
:MultiPkgName:Asynchronous Examples
```

The first way groups the Run Files and Libraries into their own menu, which is a nicer presentation to the user. The second way requires DevelopmentUtilities>Install.ctrl to be removed. Both ways would result in the same entries in the installation database.

Note that the Installation Manager installs subpackages in the order given in the various control files. If all three subpackages are chosen, they are installed in the following order: 1, Development Run Files; 2. Development Libraries; 3. Asynchronous Examples.

In this example, *Install.ctrl* is opened, and the menu contains two choices: Development Utilities and Asynchronous Examples. If the user chooses both, *DevelopmentUtilities>Install.ctrl* is opened and the menu contains the two choices Development Run Files and Development Libraries. If the user chooses both, *DevelopmentRunFiles>Install.ctrl* is opened. Since this is the bottom of the "tree," the associated command, script, and message files are copied to *[Scr]<$>* and the installation is performed. Then *DevelopmentLibraries>Install.ctrl* is opened. Again this is the bottom of the tree, so the files are copied and the installation is performed. Then *AsynchronousExamples>Install.ctrl* is opened, the files copied and that installation is performed.

If *DevelopmentRunFiles>Install.ctrl, DevelopmentLibraries>Install.ctrl,* or *AsynchronousExamples>Install.ctrl* had not been at the bottom of the tree (in other words, they contained *:MultiPkgName:* entries), menus would have been presented to the user at the time the files were opened. The subpackages listed in *DevelopmentRunFiles>Install.ctrl* would be installed before those listed in *DevelopmentLibraries>Install.ctrl*, and so on. In theory, this can go on for an infinite number of layers.

DevelopmentRunFiles>Install.ctrl File

```
:PackageName:Development Run Files
:Version:12.3.0
:Verify:
:RequiredDiskSpace:1400
:Directory:<Sys>
:Commands:'Assemble' 'LinkV6' 'Dump'
:ProductFiles:Assembler.run Linker.run Dump.run
```

DevelopmentRunFiles>InstallMsg.bin File

```
:1: "Development Package disk 2 of 3"
:2: "Development Package disk 3 of 3"
:3: "Development Libraries"
:4: "Asynchronous Examples"
:5: "Merging commands... "
:6: "Copying files... "
:7: "done."
```

DevelopmentRunFiles>Install.cmds File

The **Command File Editor** is used to create the three commands listed in the control file.

DevelopmentRunFiles>Install.jcl File

Number	Script

```
        ; DevelopmentRunFiles>Install.jcl
        ; Install the Development Run Files subpackage

        ; Except for yes/no strings, this message is the
        ; only nationalizable string in this file.

1       ErrorMsg    = 'Cannot access message file'

2       FromFloppy  = 0
        FromServer  = 2
3       FloppySpec  = ConcatStrings(DeviceFrom,'<CTOS>')
        FromSpec    = ConcatStrings(FloppySpec, '*')
        ToSpec      = ConcatStrings(VolumeTo, DirectoryTo)

4       InitErc = InitMsgFile(MsgFile)
        If InitErc > 0
            DisplayLine(ErrorMsg)
            Cancel
        Endif

5       CopyMsg = GetMsg(6)
        DoneMsg = GetMsg(7)
        Msg = GetMsg(5)
        Display(Msg)

6       Command Merge Command Files ,&
            CmdFileFrom             ,&
            CmdFileTo               ,&
            Yes                     ,&
            VolumeTo

        Displayline(DoneMsg)
        DisplayLine
        Display(CopyMsg)
```

Number	Comment
	When values for installation variables are given, it is assumed that the user has not used the *Examine / Change Defaults* option from the Installation Defaults menu. For floppy disk installations, it is assumed that the user has inserted the floppy disk in drive *[f0].*
1.	Assign strings to be used later.
2.	Assign values to enhance readability.
3.	*DeviceFrom* is *[f0]* if the installation is private, [!Sys] if the installation is public. *VolumeTo* is *[Sys]* if the installation is private, *[!Sys]* if public. *DirectoryTo* is *<Sys>*.
4.	Initialize the message file. The message file is *[Scr]<$>DevelopmentRunFiles>InstallMsg.bin* if the installation is private or *[!Sys]<Installed> DevelopmentRunFiles>12.3.0>InstallMsg.bin* if the installation is from the server. Print a message if the file does not exist. It should always be there; this is just defensive coding. The Cancel statement causes termination of script processing by Batch. The installation will fail because the product files were not updated.
5.	Since the "copying" and "done" messages are used more than once, put them in their own variables.
6.	Merge the command files. *CmdFileFrom* is *[Scr]<$>DevelopmentRunFiles>Install.cmds* if the installation is private or *[!Sys]<Installed>DevelopmentRunFiles>12.3.0>Install.cmds* if the installation is from the server. *CmdFileTo* is *[Sys]<Sys>Sys.cmds* if the installation is private, *[!Sys]<Sys>Cluster.cmds* if it is public. *VolumeTo* is *[Sys]* if the installation is private, *[!Sys]* if it is public.

Number	Script
7	If InstallType = FromServer
	Command LCopy ,&
	,&
	@[Scr]<$>InstallOverwriteOkFrom.fls,&
	@[Scr]<$>InstallOverwriteOkTo.fls,&
	,&
	Yes ,&
	No ,&
	No ,&
	Yes
	Displayline(DoneMsg)
	DisplayLine
8	GoTo Endit
	Endif
	RestartLabel
9	Command LCopy ,&
	FromSpec ,&
	,&
	ToSpec ,&
	,&
	Yes ,&
	No ,&
	No ,&
	Yes
	Displayline(DoneMsg)
	DisplayLine
	RestartLabel
10	Msg = GetMsg(1)
	NextFloppy(Msg)
	DisplayLine
	Display(CopyMsg)

Number	Comment
7.	When the user does a public installation, this script file, along with the accompanying command and message files is copied to *[!Sys]<Installed>* with the prefix *DevelopmentRunFiles>12.3.0>*. The script is invoked when the user attempts to install the product from the server. Therefore it (and indeed every script) must allow for the case when the user installs from the server (even if the script displays a message saying that installation from the server is not supported for the subpackage). This **LCopy** command is performed if the user invoked the Installation Manager using the Executive command **Server Install** or if the *Install from server* option was chosen from the Installation Media menu. *[Scr]<$>InstallOverwriteOkFrom.fls* contains: *[!Sys]<Sys>Assembler.run* *[!Sys]<Sys>Linker.run* *[!Sys]<Sys>Dump.run* *[Scr]<$>InstallOverwriteOkTo.fls* contains: *[Sys]<Sys>Assembler.run* *[Sys]<Sys>Linker.run* *[Sys]<Sys>Dump.run*
8.	Since there is nothing more to do in the case of an installation from the server, quit. A GoTo statement is used to make the script more readable. An Else clause would have been equally correct.
9.	Everything from this point is executed only if the installation is being done from floppy disks. Copy the files from the first disk. *FromSpec* is *[f0]<CTOS>** and *ToSpec* is *[Sys]<Sys>* if the installation is private or *[!Sys]<Sys>* if the installation is public.
10.	Prompt the user to insert the next floppy disk.

Number	Script

```
11      Command LCopy                   ,&
            FromSpec                    ,&
                                        ,&
            ToSpec                      ,&
                                        ,&
                Yes                     ,&
                No                      ,&
                No                      ,&
                Yes

        Displayline(DoneMsg)
        DisplayLine

12      Pkg = GetMsg(3)
        If SubString(Pkg,Pkgs) <> 65535
            Msg = GetMsg(1)
            NextFloppy(Msg)
            GoTo Endit
        Endif

13      Pkg = GetMsg(4)
        If SubString(Pkg,Pkgs) <> 65535
            Msg = GetMsg(2)
            NextFloppy(Msg)
        Endif

        Endit:
        End
```

Number	Comment
11.	Copy the files from the second disk. *FromSpec* and *ToSpec* remain the same.
12.	Determine whether the user has chosen the Development Libraries subpackage. If it was chosen, NextFloppy does not ask the user to insert a floppy disk, since it is already inserted. The statement is here so that in case the subpackage grows, minimal rewriting will have to be done to the script. The GoTo is required in case the user also chose the Asynchronous Examples subpackage. (If the GoTo were missing and the Asynchronous Examples subpackage were chosen, the user would be prompted to insert the third disk. This would cause the Installation Manager to ask the user to mount the first disk of the Development Libraries subpackage, because it would be looking for *[f0]<Sys>DevelopmentLibraries>Install.ctrl*, which is on the second disk.)
13.	If the user did not choose the Development Libraries subpackage, ascertain whether the Asynchronous Examples subpackage was chosen. If so, prompt the user to insert the third disk.

DevelopmentLibraries>Install.ctrl File

```
:PackageName:Development Libraries
:Version:12.3.0
:Verify:
:RequiredDiskSpace:800
:Directory:<Sys>
:ProductFiles:Ctos.lib Enls.lib Async.lib
```

DevelopmentLibraries>InstallMsg.bin File

```
:1: "Development Package disk 3 of 3"
:2: "Copying files... "
:3: "done."
```

DevelopmentLibraries>Install.jcl File

Number	Script

```
        ; DevelopmentLibraries>Install.jcl
        ; Install the Development Libraries subpackage

        ; Except for yes/no strings, these two messages
        ; are the only nationalizable strings in this
        ;,file.

1       ErrorMsg    ='Cannot access message file'
        FirstDisk   ='Development Package diskette 2 of 3'

2       FromFloppy  = 0
        FromServer  = 2
3       FloppySpec  = ConcatStrings(DeviceFrom, '<Lib>')
        FromSpec    = ConcatStrings(FloppySpec, '*')
        ToSpec      = ConcatStrings(VolumeTo, DirectoryTo)

4       If InstallType = FromFloppy
            NextFloppy(FirstDisk)
        Endif

5       InitErc = InitMsgFile(MsgFile)
        If InitErc > 0
            DisplayLine(ErrorMsg)
            Cancel
        Endif

6       CopyMsg = GetMsg(2)
        DoneMsg = GetMsg(3)
        Display(CopyMsg)
```

Number	Comment
	When values for installation variables are given, it is assumed that the user has not used the *Examine/Change Defaults* option from the Installation Defaults menu. For floppy disk installations, it is assumed that the user has inserted the floppy disk in drive *[f0]*.
1.	Assign strings to be used later.
2.	Assign values to enhance readability.
3.	*DeviceFrom* is *[f0]* if the installation is private, *[!Sys]* if the installation is public. *VolumeTo* is *[Sys]* if the installation is private, *[!Sys]* if public. *DirectoryTo* is *<Sys>*.
4.	Make sure that the correct floppy disk is inserted. If the user did not choose "Development Run Files," then the first disk is still inserted.
5.	Initialize the message file. The message file is *[Scr]<$>DevelopmentLibraries>InstallMsg.bin* if the installation is private or *[!Sys]<Installed> DevelopmentLibraries>12.3.0>InstallMsg.bin* if the installation is from the server. Print a message if the file does not exist. It should always be there; this is just defensive coding. The Cancel statement causes termination of script processing by Batch. The installation will fail because the product files were not updated.
6.	Since the "copying" and "done" messages are used more than once, put them in their own variables.

Number	Script
7	If InstallType = FromServer
	Command LCopy ,&
	,&
	@[Scr]<$>InstallOverwriteOkFrom.fls,&
	@[Scr]<$>InstallOverwriteOkTo.fls ,&
	,&
	Yes ,&
	No ,&
	No ,&
	Yes
	Displayline(DoneMsg)
	DisplayLine
8	GoTo Endit
	Endif
	RestartLabel
9	Command LCopy ,&
	FromSpec ,&
	,&
	ToSpec ,&
	,&
	Yes ,&
	No ,&
	No ,&
	Yes
	Displayline(DoneMsg)
	DisplayLine
	RestartLabel
10	Msg = GetMsg(1)
	NextFloppy(Msg)
	DisplayLine
	Display(CopyMsg)

Number	Comment
7.	When the user does a public installation, this script file, along with the accompanying command and message files are copied to *[!Sys]<Installed>* with the prefix *DevelopmentLibraries>12.3.0>*. The script is invoked when the user attempts to install the product from the server. Therefore it (and indeed every script) must allow for the case when the user installs from the server (even if the script displays a message saying that installation from the server is not supported for the subpackage). This **LCopy** command is performed if the user invoked the Installation Manager using the Executive command **Server Install** or if the "Install from server" option was chosen from the Installation Media menu. *[Scr]<$>InstallOverwriteOkFrom.fls* contains: *[!Sys]<Sys>Ctos.lib* *[!Sys]<Sys>Enls.lib* *[!Sys]<Sys>Async.lib* *[Scr]<$>InstallOverwriteOkTo.fls* contains: *[Sys]<Sys>Ctos.lib* *[Sys]<Sys>Enls.lib* *[Sys]<Sys>Async.lib*
8.	Since there is nothing more to do in the case of an installation from the server, quit. A GoTo statement is used to make the script more readable. An Else clause would have been equally correct.
9.	Everything from this point gets executed only if the installation is being done from floppy disks. Copy the files from the first disk. *FromSpec* is *[f0]<Lib>** and *ToSpec* is *[Sys]<Sys>* if the installation is private or *[!Sys]<Sys>* if the installation is public.
10.	Prompt the user to insert the next floppy disk.

Number	Script

```
11      Command LCopy                   ,&
            FromSpec                    ,&
                                        ,&
            ToSpec                      ,&
                                        ,&
                Yes                     ,&
                No                      ,&
                No                      ,&
                Yes

        Endit:
        End
```

Number	Comment
11.	Copy the files from the third disk. *FromSpec* and *ToSpec* remain the same. There is no need to see if the user has chosen the Asynchronous Examples subpackage, since the third disk is already in the disk drive.

AsynchronousExamples>Install.ctrl File

```
:PackageName:Asynchronous Examples
:Version:12.3.0
:Verify:
:RequiredDiskSpace:360
:Directory:<Async>
:Commands:Start Stop Example
:ProductFiles:Async.h AsyncService.c Deinstall.c Example.c
ExDef.h ExFunc.h ExRqblk.h ExRqLabl.asm LinkDeinstall.fls
LinkDeinstall.sub LinkExample.fls LinkExample.sub
LinkStart.fls LinkStart.sub LinkStop.fls LinkStop.sub ReadMe
RequestEx.txt Start.c Stop.c
```

AsynchronousExamples>InstallMsg.bin File

```
:1: "Listing requests... "
:2: "Making Request.sys... "
:3: "Merging commands... "
:4: "Copying files... "
:5: "done."
```

AsynchronousExamples>Install.cmds File

The **Command File Editor** was used to create the three commands listed in the control file.

AsynchronousExamples>Install.jcl File

Number	Script
	`; AsynchronousExamples>Install.jcl`
	`; Install the Asynchronous Examples subpackage`
	`; Except for yes/no strings, these two messages`
	`; are the only nationalizable strings in this`
	`; file.`
1	`ErrorMsg = 'Cannot access message file'`
	`FirstDisk ='Development Package diskette 3 of 3'`
2	`FromFloppy = 0`
	`FromServer = 2`
	`TotalSilence = 2`
3	`FloppySpec = ConcatStrings(DeviceFrom, '<Async>')`
	`FromSpec = ConcatStrings(FloppySpec, '*')`
	`ToSpec = ConcatStrings(VolumeTo, DirectoryTo)`
	`ScrSpec = '[Scr]<$>'`
	`If Public`
	`SysSpec = '[!Sys]<Sys>'`
	`Else`
	`SysSpec = '[Sys]<Sys>'`
	`Endif`
	`RqSpec = ConcatStrings(SysSpec,'Request.sys')`
	`RqListSpec = ConcatStrings(ScrSpec,'LRS')`
	`RqTextSpec = ConcatStrings(ToSpec,`
	`'Request.Ex.txt')`
4	`If InstallType = FromFloppy`
	`NextFloppy(FirstDisk)`
	`Endif`
5	`InitErc = InitMsgFile(MsgFile)`
	`If InitErc > 0`
	`DisplayLine(ErrorMsg)`
	`Cancel`
	`Endif`
6	`DoneMsg = GetMsg(5)`
	`Msg = GetMsg(3)`
	`Display(Msg)`

Number	Comment
	When values for installation variables are given, it is assumed that the user has not used the *Examine / Change Defaults* option from the Installation Defaults menu. For floppy disk installations, it is assumed that the user has inserted the floppy disk in drive *[f0]*.
1.	Assign strings to be used later.
2.	Assign values to enhance readability.
3.	*DeviceFrom* is *[f0]* if the installation is private, *[!Sys]* if the installation is public. *VolumeTo* is *[Sys]* if the installation is private, *[!Sys]* if public. *DirectoryTo* is <Sys>. Note that instead of using *VolumeTo* as the volume part of *SysSpec*, we explicitly set it to *[Sys]<Sys>* or *[!Sys]<Sys>*. The user has the option of changing the destination volume to something other than *[Sys]* or *[!Sys]*, but the request file is always on the *Sys* volume of the target computer.
4.	Make sure that the correct floppy disk is inserted. If the user chose only Asynchronous Examples, then the first disk is still inserted.
5.	Initialize the message file. The message file is *[Scr]<$>AsynchronousExamples>InstallMsg.bin* if the installation is private or *[!Sys]<Installed>Asynchronous Examples>12.3.0>InstallMsg.bin* if the installation is from the server. Print a message if the file does not exist. It should always be there; this is just defensive coding. The Cancel statement causes termination of script processing by Batch. The installation will fail because the product files were not updated.
6.	Since the "copying" and "done" messages are used more than once, put them in their own variables.

Number	Script

7	`Command Merge Command Files ,&`
	`  CmdFileFrom ,&`
	`  CmdFileTo ,&`
	`  Yes ,&`
	`  VolumeTo]`
	
	`Displayline(DoneMsg)`
	`DisplayLine`
	`Msg = GetMsg(4)`
	`Display(Msg)`

8	`If InstallType = FromServer`
	`  Command LCopy ,&`
	`  ,&`
	`  @[Scr]<$>InstallOverwriteOkFrom.fls,&`
	`  @[Scr]<$>InstallOverwriteOkTo.fls,&`
	`  ,&`
	`  Yes ,&`
	`  No ,&`
	`  No ,&`
	`  Yes`

Number	Comment
7.	Merge the command files. *CmdFileFrom* is *[Scr]<$>AsynchronousExamples>Install.cmds* if the installation is private or *[!Sys]<Installed> AsynchronousExamples>12.3.0>Install.cmds* if the installation is from the server. *CmdFileTo* is *[Sys]<Sys>Sys.cmds* if the installation is private, *[!Sys]<Sys>Cluster.cmds* if it is public. *VolumeTo* is *[Sys]* if the installation is private, *[!Sys]* if it is public.
8.	When the user does a public installation, this script file, along with the accompanying command and message files are copied to *[!Sys]<Installed>* with the prefix *AsynchronousExamples>12.3.0>*. The script is invoked when the user attempts to install the product from the server. Therefore it (and indeed every script) must allow for the case when the user installs from the server (even if the script displays a message saying that installation from the server is not supported for the subpackage). This **LCopy** command is performed if the user invoked the Installation Manager using the Executive command **Server Install** or if the *Install from server* option was chosen from the Installation Media menu. *[Scr]<$>InstallOverwriteOkFrom.fls* contains: *[!Sys]<Async>Async.h* *[!Sys]<Async>AsyncService.c* *[!Sys]<Async>Deinstall.c* *[!Sys]<Async>Example.c* *[!Sys]<Async>ExDef.h* *[!Sys]<Async>ExFunc.h* *[!Sys]<Async>ExRqblk.h* *[!Sys]<Async>ExRqLabl.asm* *[!Sys]<Async>LinkDeinstall.fls* *[!Sys]<Async>LinkDeinstall.sub* *[!Sys]<Async>LinkExample.fls* *[!Sys]<Async>LinkExample.sub* *[!Sys]<Async>LinkStart.fls* *[!Sys]<Async>LinkStart.sub* *List continues....*

Number	Script

```
9           GoTo DoRqs
        Endif

        RestartLabel

10      Command LCopy                   ,&
            FromSpec                    ,&
                                        ,&
            ToSpec                      ,&
                                        ,&
                Yes                     ,&
                No                      ,&
                No                      ,&
                Yes

        DoRqs:

        Displayline(DoneMsg)
        DisplayLine

        RestartLabel

        Msg = GetMsg(1)
        Display(Msg)
```

Number	Comment
	[Scr]<$>InstallOverwriteOkFrom.fls continued: *[!Sys]<Async>LinkStop.fls* *[!Sys]<Async>LinkStop.sub* *[!Sys]<Async>ReadMe* *[!Sys]<Async>RequestEx.txt* *[!Sys]<Async>Start.c* *[!Sys]<Async>Stop.c]* *[Scr]<$>InstallOverwriteOkTo.fls* will contain: *[Sys]<Async>Async.h* *[Sys]<Async>AsyncService.c* *[Sys]<Async>Deinstall.c* *[Sys]<Async>Example.c* *[Sys]<Async>ExDef.h* *[Sys]<Async>ExFunc.h* *[Sys]<Async>ExRqblk.h* *[Sys]<Async>ExRqLabl.asm* *[Sys]<Async>LinkDeinstall.fls* *[Sys]<Async>LinkDeinstall.sub* *[Sys]<Async>LinkExample.fls* *[Sys]<Async>LinkExample.sub* *[Sys]<Async>LinkStart.fls* *[Sys]<Async>LinkStart.sub* *[Sys]<Async>LinkStop.fls* *[Sys]<Async>LinkStop.sub* *[Sys]<Async>ReadMe* *[Sys]<Async>RequestEx.txt* *[Sys]<Async>Start.c* *[Sys]<Async>Stop.c*
9.	Merge the requests.
10.	Copy the files from the disk. *FromSpec* is *[f0]<Async>** and *ToSpec* is *[Sys]<Async>* if the installation is private or *[!Sys]<Async>* if the installation is public.

Number	Script

```
11      Command List Request Set     ,&
            RqSpec                   ,&
            RqListSpec]

        Displayline(DoneMsg)
        DisplayLine
        Msg = GetMsg(2)
        Display(Msg)

12      SaveVidLevel = VideoLevel
        If VideoLevel = TotalSilence
            EchoSome
        EndIf

Command Make Request Set             ,&
            (RqListSpec, RqTextSpec),&
            RqSpec                   ,&
                                     ,&
            12.3.0

If SaveVidLevel = TotalSilence
        EchoOff
  EndIf

        Displayline(DoneMsg)
        DisplayLine

        End
```

Number	Comment

11. List the request set. *RqSpec* is *[Sys]<Sys>Request.sys* if the installation is private or *[!Sys]<Sys>Request.sys*. *RqListSpec* is *[Scr]<$>LRS*.

12. Merge the new requests into the user's request file. Note that video must be turned on, since there might be errors reported by **Make Request Set**. If it reports errors, it stops for user input. If video is off (non verbose), the installation appears to hang, since the user never sees the messages. *RqListSpec* and *RqSpec* are unchanged. *RqTextSpec* is *[Sys]<Async>Request.Ex.txt* if the installation is private or *[!Sys]<Async>Request.Ex.txt* if the installation is public.

Example 3: Tape Installation

This last example is a tape installation of the second example (Development Package). In this example, the following events occur:

1. The product files are copied for each of the chosen subpackages.
2. Two of the subpackages have associated command files (Development Run Files and Asynchronous Examples). If one (or both) of these subpackages is chosen, its command files are merged.
3. One of the subpackages (Asynchronous Examples) has commands that need to be merged if that subpackage is chosen.

The string variable pkgs contains the names of all the subpackages chosen by the user. Install Manager passes this variable to Batch. The JCL installation script then calls the Batch statement Substring to query this variable.

The installation files are:

Install.ctrl (same as DevelopmentUtilities>Install.ctrl in the previous example)
Install.jcl (see following pages)
InstallMsg.bin (see following pages)

DevelopmentRunFiles>Install.ctrl
DevelopmentRunFiles>Install.cmds

DevelopmentLibraries>Install.ctrl

AsynchronousExamples>Install.ctrl
AsynchronousExamples>Install.cmds

These files are placed in *[Qic]0*.

The contents of the other installation files are the same as that of the files in the previous example.

InstallMsg.bin File

```
:1: "Listing requests... "
:2: "Making Request.sys... "
:3: "Merging commands... "
:4: "Restoring files... "
:5: "done."
:6: "Asynchronous Examples"
:7: "AsynchronousExamples>Install.cmds"
:8: "Development Run Files"
:9: "DevelopmentRunFiles>Install.cmds"
```

Install.jcl File

Number	Script
	`; Install.jcl`
	`; Install Development Package from Qic Tape`
	`; Except for yes/no strings, this message is the`
	`; only nationalizable string in this file.`
1	`ErrorMsg = 'Cannot access message file'`
2	`ScrSpec = '[Scr]<$>'`
	`ToSpec = ConcatStrings(VolumeTo, DirectoryTo)`
	`If Public`
	`  SysSpec = '[!Sys]<Sys>'`
	`Else`
	`  SysSpec = '[Sys]<Sys>'`
	`Endif`
	`RqSpec = ConcatStrings(SysSpec,'Request.sys')`
	`RqListSpec = ConcatStrings(ScrSpec, 'LRS')`
	`RqTextSpec = ConcatStrings(ToSpec,`
	`  'Request.vm.txt')`
3	`InitErc = InitMsgFile(MsgFile)`
	`If InitErc > 0`
	`  DisplayLine(ErrorMsg)`
	`  Cancel`
	`Endif`
4	`DoneMsg = GetMsg(5)`
	`Msg = GetMsg(4)`
	`Display(Msg)`
	`RestartLabel`

Number	Comment
	When values for installation variables are given, it is assumed that the user has not used the *Examine/Change Defaults* option from the Installation Defaults menu. Note that in this script, we don't have to worry about floppy and server installations, since the script file is used only for tape installations.
1.	Assign strings to be used later.
2.	*VolumeTo* is *[Sys]* if the installation is private, *[!Sys]* if public. *DirectoryTo* is *<Sys>*. Note that instead of using *VolumeTo* as the volume part of SysSpec, we explicitly set it to *[Sys]<Sys>* or *[!Sys]<Sys>*. The user has the option of changing the destination volume to something other than *[Sys]* or *[!Sys]*, but the request file is always always on the *Sys* volume of the target computer.
3.	Initialize the message file. The message file is *[Scr]<$>InstallMsg.bin*. Print a message if the file does not exist. It should always be there; this is just defensive coding. The Cancel statement causes termination of script processing by Batch. The installation will fail because the product files were not updated.
4.	Since the "copying" and "done" messages are used more than once, put them in their own variables.

Number	Script

5	`Command Restore Archive ,&`
	`DeviceFrom ,&`
	`@[Scr]<$>InstallOverwriteOkFrom.fls,&`
	`@[Scr]<$>InstallOverwriteOkTo.fls,&`
	`yes`
	
	`Displayline(DoneMsg)`
	`DisplayLine`
	
	`RestartLabel`

Number	Comment
5.	Restore the product files. We will assume that the user chose all three subpackages. Since **Restore Archive** will create directories that do not already exist, we do not have to create *[VolumeTo]<Async>*. *DeviceFrom* is *[Qic]1*. *[Scr]<$>InstallOverwriteOkFrom.fls* contains: *<*>Assembler.run* *<*>Linker.run* *<*>Dump.run* *<*>Ctos.lib* *<*>Enls.lib* *<*>Async.lib* *<Async>Async.h* *<*>AsyncService.c* *<*>Deinstall.c* *<*>Example.c* *<*>ExDef.h* *<*>ExFunc.h* *<*>ExRqblk.h* *<*>ExRqLabl.asm* *<*>LinkDeinstall.fls* *<*>LinkDeinstall.sub* *<*>LinkExample.fls* *<*>LinkExample.sub* *<*>LinkStart.fls* *<*>LinkStart.sub* *<*>LinkStop.fls* *<*>LinkStop.sub* *<*>ReadMe* *<*>RequestEx.txt* *<*>Start.c* *<*>Stop.c*

[Scr]<$>InstallOverwriteOkTo.fls will contain:

[Sys]<Sys>Assembler.run
[Sys]<Sys>Linker.run
[Sys]<Sys>Dump.run
[Sys]<Sys>Ctos.lib
[Sys]<Sys>Enls.lib
[Sys]<Sys>Async.lib
[Sys]<Async>Async.h
[Sys]<Async>AsyncService.c
[Sys]<Async>Deinstall.c
[Sys]<Async>Example.c
[Sys]<Async>ExDef.h
[Sys]<Async>ExFunc.h
[Sys]<Async>ExRqblk.h
[Sys]<Async>ExRqLabl.asm
[Sys]<Async>LinkDeinstall.fls
[Sys]<Async>LinkDeinstall.sub
[Sys]<Async>LinkExample.fls
[Sys]<Async>LinkExample.sub
[Sys]<Async>LinkStart.fls
[Sys]<Async>LinkStart.sub
[Sys]<Async>LinkStop.fls
[Sys]<Async>LinkStop.sub
[Sys]<Async>ReadMe
[Sys]<Async>RequestEx.txt
[Sys]<Async>Start.c
[Sys]<Async>Stop.c

If at least one of the control files contained an entry for *OverwriteNoFiles,* those would also have to be restored by the statements:

```
OverwriteNo=ConcatStrings(ScrSpec,
       'InstallOverwriteNoFrom.fls')
Ver = FileVersion(OverwriteNo)
If Ver = '255' Then
   Command Restore Archive                          ,&
           DeviceFrom                               ,&
           @[Scr]<$>InstallOverwriteNoFrom.fls      ,&
           @[Scr]<$>InstallOverwriteNoTo.fls        ,&
           No
EndIf
```

If there are no entries for *OverwriteNoFiles* in the control files of the chosen subpackages, *InstallOverwriteNoFrom.fls* and *InstallOverwriteNoTo.fls* are not created. Therefore, we have to check for their existence. A value of '255' from FileVersion() says that the file exists but doesn't have a version (only run files have versions). If the files did not exist and the **Restore Archive** command was issued anyway, the second and third parameters would be null. That means that the default of <*>* would be used, which is clearly not what is desired.

Number	Script

```
6       pkg = GetMsg(6)
        If substring(pkg,pkgs) <> 65535
            Msg = GetMsg(1)
            Display(Msg)

7           Command List Request Set,&
                      RqSpec        ,&
                      RqListSpec

            Displayline(DoneMsg)
            DisplayLine
            Msg = GetMsg(2)
            Display(Msg)

8       SaveVidLevel = VideoLevel
        If VideoLevel = TotalSilence
            EchoSome
        EndIf

        Command Make Request Set     ,&
                     (RqListSpec, RqTextSpec),&
                     RqSpec         ,&
                                    ,&
                     12.3.0

If SaveVidLevel = TotalSilence
        EchoOff
  EndIf

            Displayline(DoneMsg)
            DisplayLine
```

Number	Comment
6.	Determine whether the Asynchronous Examples subpackage has been chosen. If so, print the message "Listing requests . ." Then perform steps 7, 8, and 9 (**List Request Set, Make Request Set, Merge Command Files**).
7.	List the request set: *RqSpec is [Sys]<Sys>Request.sys* if the installation is private or *[!Sys]<Sys>Request.sys* if the installation is public. *RqListSpec* is *[Scr]<$>LRS*.
8.	Merge the new requests into the user's request file. Note that video must be turned on, since there might be errors reported by **Make Request Set**. If it reports errors, it stops for user input. If video is off (non verbose), the installation appears to hang, since the user never sees the messages. *RqListSpec* and *RqSpec* are unchanged. *RqTextSpec* is *[Sys]<Async>Request.Ex.txt* if the installation is private or *[!Sys]<Async>Request.Ex.txt* if the installation is public.

Number	Script
9	pkgCmdName = GetMsg(7) CmdSpec = ConcatStrings(ScrSpec, pkgCmdName) Command Merge Command Files,& CmdSpec ,& CmdFileTo ,& Yes ,& VolumeTo EndIf RestartLabel
10	pkg = GetMsg(8) If substring(pkg,pkgs) <> 65535 pkgCmdName = GetMsg(9) CmdSpec = ConcatStrings(ScrSpec, pkgCmdName) Command Merge Command Files,& CmdSpec ,& CmdFileTo ,& Yes ,& VolumeTo EndIf End

Number	Comment
9.	Merge the command files. *CmdFileTo* is *[Sys]<Sys>Sys.cmds* if the installation is private, *[!Sys]<Sys>Cluster.cmds* if it is public. *VolumeTo* is *[Sys]* if the installation is private, *[!Sys]* if it is public. Note that the names of the command files must come from the message file, since there is no way for the Installation Manager to pass them all to Batch.
10.	If the Development Run Files subpackage was chosen, merge its commands.

Section 7
Using the System Log File

Introduction

The system log file, *Log.sys*, is created when you initialize a disk with the **Format Disk** or **Configure Disk** command. Any application can read or write to the log file, but it is used primarily by the operating system and by system services to log significant events. For example, the file system logs all disk errors in the system log file. End users can view the log file's contents by using the Executive's **PLog** command. For details on **Format Disk**, **Configure Disk**, and **PLog**, see the *CTOS Executive Reference Manual.*

Only certain types of applications would find the log file useful. For example, a system monitor program might check the log file periodically and send mail to a remote administrator if it finds any error messages there.

This section describes the internal format of the log file and explains how applications can use it. The section assumes that you understand how to use the file management operations. For background, read "File Management," in the *CTOS Operating System Concepts Manual.*

Log File Format

The contents of the system log file are called records. There are a number of different record types for the display of different kinds of information. Each record type, however, has the same 31-byte header format, which is automatically generated by the operating system. One of the fields in the header contains the record size. This field is mentioned several times in this section because of its significance in reading the log file. The record trailer is the type-specific information of the record. The length and organization of the trailer varies with the type of record. The log-file format as well as the different record types are described in detail in "System Structures," in the *CTOS Procedural Interface Reference Manual.* To read the log file, you should be familiar with these formats.

The size the log file is set at volume initialization and can be changed only by reinitializing the disk. For this reason, if the file gets full, the operating system writes new records in circular fashion, starting over again at the beginning of the first file sector. In this way, when the file becomes full, the oldest information is overwritten with the newest information.

Writing Messages to the Log File

The operating system and utilities write different types of records to *Log.sys*. The application program, however, can write only an ASCII record type, and this must be done using the WriteLog request. The file system is the only program that writes to the log file. It acts as the intermediary by serving WriteLog requests and doing the writing to the log file. For an example of how to use WriteLog, see "Error Handling Conventions," in *CTOS / Open Programming Practices and Standards.* The WriteLog request itself is described in detail in the *CTOS Procedural Interface Reference Manual.*

Displaying Messages Using PLog

If you use **PLog** to display text messages that have been entered in the log file with WriteLog, **PLog** displays the header information followed by:

```
ASCII MESSAGE
```

Following this information, the contents of your message are displayed. Because **PLog** uses video byte streams (calls WriteBsRecord) to display messages, it does not format messages.

Although you can use **PLog** manually, you can also write your own routine to display the contents of the log file. To do so, however, requires that you know how to read the file. Reading the file, in turn, makes more sense if you understand how the file system writes to it in the first place. Finally, to perform any of these tasks, you need to be familiar with a few fields in the Volume Home Block (VHB). Let's look at the structures first.

Log File Fields in the Volume Home Block

Certain fields in the Volume Home Block (VHB) are used to read from or write to the log file. These are defined below:

lfaLogBase	Is the disk location where the log file is placed when the file is created. This field is important only to the file system, which has many requests to access the disk and needs to make these requests as rapidly as possible.
cPageLog	Is the number of disk sectors in the log file.
currentLogPage	Is the number of the current sector to be written to (a value in the range 0 through *cPageLog*-1).
currentLogByte	Is the next byte to be written to in the current sector (a value in the range 0 through 511).

lfaLogBase and *cPageLog* are static values determined at disk initialization. *currentLogPage* and *currentLogByte*, however, are values that change dynamically as the file system writes to the log file.

How the File System Writes to the Log File

The file system always writes to the current sector (designated by the value of *currentLogPage*). First it reads the contents of the current (512-byte) sector into a 512-byte buffer in memory. Because there may be valid records already written to the sector, it then compares the combined value of *currentLogByte* and the record size with 512. If the combined value is less than or equal to 512, the new record will fit in the buffer.

Record Fits in the Buffer

If the record fits in the buffer, the file system

1. Copies the new record to the buffer starting at offset *currentLogByte*.
2. Writes the buffer to the disk sector specified by *currentLogPage*.
3. Increments the value of *currentLogByte* by the record size.

Record Does Not Fit in the Buffer

If the record doesn't fit in the buffer, the file system

1. Increments the value of *currentLogPage* by 1.
2. Initializes the buffer to zeros.
3. Places the record at the beginning of the buffer.
4. Writes the buffer to the disk sector specified by the new *currentLogPage*.
5. Sets the value of *currentLogByte* to the record size.

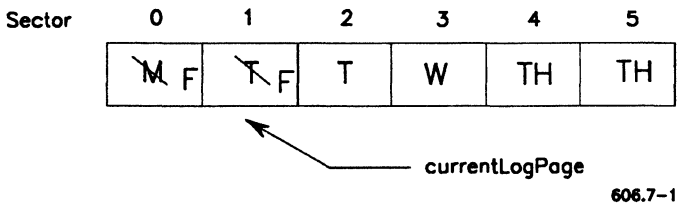
Incrementing *currentLogPage* by 1 means that, when the buffer is written back to disk, it is written to the next file sector. This preserves the contents of the sector that didn't have enough room for the record. Initializing the buffer to zeros not only eliminates garbage that may have been in memory but also indicates the last valid record in a sector to the log file reader. (The significance of zeroing out the buffer to find the last record is explained in "How to Read the log file," later in this section.)

If the new value of *currentLogPage* is greater than or equal to the number of sectors in the log file (*cPageLog*), the file system sets *currentLogPage* to 0 before it writes the buffer to disk. When this condition occurs, the file system OR's the value of *currentLogByte* with the mask 8000h. For example, if under these circumstances *currentLogPage* is 000Ch, the file system OR's this value to 800Ch. Doing so indicates to a program reading the log file that the records have wrapped around to the beginning of the file: the earliest chronological sector is not the first log file sector on the disk.

Figure 7-1 shows the contents of a 6-sector log file that has wrapped around. The letters M, T, W, TH, and F represent the days of the week to which the corresponding sectors shown were written. Sector 0 was written on Monday. On Tuesday sectors 1 and 2 were written, and so forth. Note that Thursday's input filled up the last sector in the file. Because of this, the file system had to write Friday's input to sectors 0 and 1. When the file system wrote to sector 0, it reset *currentLogPage* to 0 and OR'd the value of *currentLogByte* with 8000h for the benefit of the log file reader.

The figure shows that sector 2 contains the oldest information. This sector is now the *chronological* beginning of the file even though sector 0 at LFA 0 is still the beginning of the first file sector. The chronological beginning of this file starts at the beginning of the third sector or LFA (2*512).

Figure 7-1. Log File Wraparound



How to Read the Log File

Any program can read the log file. A system service might be written, for example, to read the log file on an occasional basis, selectively looking for errors that may need to be called to the attention of a system administrator. At the request of the user, *PLog.run* reads and displays the contents of the entire file.

Before reading the log file, your program needs to gain access to the log file fields in the VHB. Then it needs to determine whether the records in the file have wrapped around.

Accessing the Log-File Fields in the VHB

Before reading the log file, your program needs to gain access to the following fields in the VHB: *cPageLog*, *currentLogPage*, and *currentLogByte*. To do so, your program can call the GetpStructure operation to get a pointer to the VHB. (GetpStructure is described in detail in the *CTOS Procedural Interface Reference Manual.*)

Determining if Records Have Wrapped Around

To find out whether the records have wrapped around so that your program can tell where to start and stop reading, AND the value of *currentLogByte* with the mask 8000h. A nonzero value means the records have wrapped around, for example, ANDing with a *currentLogByte* value of 800Ch is shown below:

1000 0000 0000 0000	8000h mask
1000 0000 0000 1100	800Ch *currentLogByte*
1000 0000 0000 0000	Result of ANDing is a nonzero value

Reading the File in Chronological Order

How you read the log file records in chronological order depends on whether the records have wrapped around.

If Records Have Not Wrapped Around

If the records have not wrapped around (*currentLogByte* AND 8000h equals zero), start reading the first file sector at LFA 0 and proceed as follows:

1. Read the first record starting at offset 0 in the sector. Then process the record. (*Process* means do what you want with it. Depending on the record type, you may wish to skip the record, print it, and so forth. You can obtain general information about the record, such as its type and size from the 31-byte standard record header automatically generated when the file system wrote the record to the log file.)
2. To read the next record, increment the offset by the size of the first record. Then check the field *Record.size* in the next record's header. If this value is 0, you have reached the end of the valid records in this sector. This is so because when the file system sees that the next record to be written won't fit in the current sector, it writes the record to the next sector using a buffer initialized with zeros. As a result, the unused portion of a sector always contains zero values. (See "How the File System Writes to the Log File," earlier in this section.)
3. Continue processing records and incrementing the offset to advance to the next record until the value of *Record.size* equals 0.
4. Compare the number of sectors you've read so far to the value of (*currentLogPage*+1). If the numbers are equal, you are done reading all the records in the log file. Otherwise, repeat steps 1 through 4 with the next sector until you have processed *currentLogPage*+1 sectors.

If Records Have Wrapped Around

If the records in the file have wrapped around (*currentLogByte* AND 8000h equals a nonzero value), proceed as follows:

1. To start reading from the earliest sector written, read the file beginning at sector (*currentLogPage*+1). (See Figure 7-1. In the figure, *currentLogPage* is 1. The earliest sector written is sector 2.)
2. Process the records in this sector as described in steps 1 through 3 in "If Records Have Not Wrapped Around".
3. If this sector is the last file sector [(*currentLogPage*+1) = *cPageLog*], proceed to step 4. Otherwise, continue processing the records in each sector until you have processed all records in the last file sector.
4. At the end of the last file sector, read the file starting at the beginning of the first file sector (LFA 0). Process the records as described in "If Records Have Not Wrapped Around".

Reading the File in Reverse Chronological Order

Reading the file in reverse chronological order requires you to determine the start of each record. You must also deal with record wrap around.

Saving Offsets to Records

Reverse chronological order is the order **PLog** uses to read and display files.

Because the records in the log file are of variable length, the only way to find out where each record starts in a sector is to read the sector first from the beginning to the end. Record the size of each record you read. You will know when you get to the last record in the sector because *Record.size* will be 0. From the record sizes, calculate the offsets to the beginning of each record, and save these values in an array.

The next two sections present a step-by-step intuitive approach to reading the records. This is followed by the algorithm **PLog** uses to process each sector.

If Records Have Not Wrapped Around

1. Start at the sector specified by *currentLogPage*.
2. Save away record offsets as described in "Saving Offsets to Records,".
3. Look up and process each record in the array of record offsets, starting with the record at the last offset in the sector and proceeding to the record at offset 0.
4. Repeat steps 2 and 3 for the sector preceding the current one (*currentLogPage*-1).
5. Repeat step 4 until you have read the first sector in the file.

If Records Have Wrapped Around

1. Perform all steps described in "If Records Have Not Wrapped Around," above.
2. Start with the last sector in the file. (The value of *cPageLog* is the number of this sector).
3. Repeat steps 2 through 4 in "If Records Have Not Wrapped Around," above, until you have read the sector number (*currentLogPage*+1).

PLog's Algorithm For Processing Each Sector

To process each sector, **PLog** uses the algorithm shown below. The algorithm is shown in pseudocode. It is not intended to be compiled. The variables in the algorithm have the following meanings:

Buffer	Is a 512-byte array.
Lfa	Is the value (512 * page number currently being processed).
pRecord	Points to a Log File Record structure. The format of this structure is described in "Log File Record Format" in "System Structures" in the *CTOS Procedural Interface Reference Manual.*
pTRecord	Points to another Log File Record structure. The format of this structure is the same as *Record.*

This algorithm is used once for each sector of the log file.

Example 7-1. PLog's Record-Processing Algorithm

```
/*  Previous Declarations
    LogFileRecordType  *pRecord
    LogFileRecordType  *pTRecord
*/

erc = Read(fhLogFile,&Buffer,512,Lfa,&cb);
pRecord = &Buffer;
pTRecord = &Buffer;

do while (pRecord->size != 0) {
    i = 0;
    do while ((pTRecord->size != 0) && (i < cb)) {
                                          pRecord = pTRecord;
                                        i = i + Record.size;
                                      pTRecord = &Buffer[i];
                                                          }
    ProcessRecord(pRecord);
    pRecord->size = 0;
    }
```

Section 8
Writing Multiprocessor System Services

Introduction

This section describes some of the issues you need to resolve if you want to write system services for multiprocessor systems. The multiprocessor systems this section specifically addresses are the following protected-mode operating systems and hardware:

- CTOS III 1.1 or greater, on an EISA-bus workstation with at least one CTOS I/O Processor Card
- CTOS/XE 3.0 or greater, on an XE-530

This section addresses some common pitfalls encountered when porting system services from earlier hardware and operating system versions. It also describes some recommended techniques to help make new system services compatible with current and future products.

In general, this section calls the XE-530 a shared resource processor (SRP).

Portation Issues for Existing Programs

This subsection starts out by describing some of the basic issues you may need to address if you plan to convert an existing real-mode system service to run on a multiprocessor system.

General Guidelines

CTOS III 1.1 and CTOS/XE 3.0 are protected-mode operating systems. For a system service to run on a protected-mode system, it must execute in protected mode; real-mode system services are not supported. By conforming to the general guidelines below, most system services and general applications (for that matter), should be able to run in protected mode.

As a first general guideline, your system service must conform to the general guidelines for protected-mode programming. (For details on these guidelines, see "Protected Mode Programming Guidelines," in *CTOS/Open Programming Practices and Standards.*) If your service conforms to the guidelines in that section, relinking may be all that is required to run it in protected mode.

Secondly, your system service must not contain direct memory references to operating system data structures. On protected-mode systems, your system service cannot depend on the addresses of such structures. There is no guarantee they will be in the same locations they occupied on real-mode versions of the operating system. If your service must access a structure, it should do so by calling the GetPStructure operation with the appropriate case value for the structure.

Unsupported Mechanisms

The subsections below describe mechanisms some real-mode applications and system services use illicitly. These mechanisms are not unsupported on protected-mode multiprocessor operating systems.

User Numbers and Exchanges

Real-mode operating systems assigned exchanges and user numbers in the range 1 to 255. Some programs took advantage of this fact, using the high-order byte of the *UserNum* and/or *ExchResp* fields of a request block for their own purposes.

Multiprocessor systems use the full word size of the user number and the response exchange fields. Any programs using these fields to contain values other than the true user number or response exchange will fail.

Remote Memory and Inter-CPU Communication

On earlier versions of SRP operating systems it was possible (though not recommended) for a program to bypass the inter-CPU communication (ICC) mechanism and write directly to the memory of a remote processor. This is not possible on a protected-mode multiprocessor system.

The internal mechanism by which processors communicate has changed substantially, and existing programs that attempt to use the old mechanism will fail.

If a program needs to communicate with a remote processor, it should use the operating system Request interface. The performance of the Request interface has been substantially improved over earlier operating system versions, even on existing hardware. As such, there is no reason programs should need to bypass it.

Restricted Use of GetWSUserName on SRPs

On workstation servers, the GetWSUserName operation can be used to obtain the user name of any cluster user. On SRPs, however, GetWSUserName should not be used. At best, it retrieves only those user names connected to the local processor. There is no clean method of identifying the users of a system service on an SRP.

Controlling Interprocessor Request Routing

This subsection describes the interprocessor request routing methods supported on multiprocessor systems. System services must define some form of interprocessor routing for the requests they serve. The text below describes each of the routing methods available and makes recommendations on using these methods.

Note: *The* Request.txt *definition file entry for interprocessor request routing is* :SrpRouting:. *Historically this routing type was only available on shared resource processors (SRPs) to route requests between SRP processor boards. With the introduction of CTOS III 1.1 on EISA-bus multiprocessor workstations, you use this same request-file entry to define request routing between CTOS I/O Processor Cards. Other documentation may call interprocessor routing* SRP request routing.

Interprocessor Request Routing Types

Table 8-1 lists and briefly describes the interprocessor request routing types.

The text that follows describes the three most commonly used interprocessor routing types in greater detail. Almost all system services should use one of these. The other types are intended for rare special cases and for internal use by the operating system.

Table 8-1. Interprocessor Request Routing Types

Value	Description
rDevice*	If the request definition does not specify either a handle or a specification (that is, no network routing), the request is routed to all processors previously accessed by this user number. An example of such a request is a termination request. If the request definition specifies a resource handle (network routing by handle), it is routed by the handle to the processor that allocated the handle. If the request definition includes a specification type such as FileSpec or DirSpec for the network routing parameter, the operating system kernel searches for the device name in square brackets (such as *[GP02]*) in the CPU device name table. The table entry associates the device name with the slot number of the processor where the request is served. (If the specification does not contain a device or volume name, the kernel attempts to send the request to a local system service.)
rRemote*	If the request is served locally, rRemote is the same as rLocal. Otherwise the kernel sends the request by exchange to the remote processor where the service is located. When a system service calls ServeRq during installation, ServeRq updates the operating system request routing table on each processor with the system service exchange. The exchange uniquely identifies the processor upon which the system service is running. As such, only one instance of a system service can serve a request routed rRemote on a multiprocessor system.

*This type is frequently used.

continued

Table 8-1. Interprocessor Request Routing Types (cont.)

Field	Description
rLocal*	If the request definition does not include a specification type such as FileSpec or DirSpec for a remote processor, the request is served locally. The service exchange is looked up in the service exchange field of the operating system request routing table. If the request contains a specification for a remote processor, the operating system kernel looks up the device name in the local CPU device name table to determine which slot to send the request to, and it routes the request to that processor.
rBroadcast	The request is routed to every processor booted in the system.
rUserId	The request is routed to the processor owning the user number contained in the first word of the request block control information. Information encoded in the user number uniquely identifies the target processor.
rMasterFp	The request is routed to the master processor (processor that boots all others).
rFileId	The request is routed to the appropriate processor by the slot number contained in the first byte of the request block control information.
rLineNum	The request is routed to the cluster processor (Cluster I/O Processor card, General Processor or Cluster Processor board) that handles this line. This operation is used by the GetClusterStatus operation.
rHandle	The request is routed to the target file processor (such as a GP+SI on an SRP or the master processor on an EISA-bus workstation) using an indexed field in the handle.
rMasterCp	(Unused)

*This type is frequently used.

Routing by Device Specification

Routing by device specification (rDevice) enables you to take advantage of installing multiple instances of the same system service on different processors. In cases where a system service is widely used, device specification routing is the recommended routing type to use.

For example, this routing type allows an SNA Network Gateway to be installed on each of two GP+CI boards. A system service client process running on a client workstation can then specify which of the SNA Network Gateways it wants to use, by naming the processor board on which that Gateway is installed.

Routing by device specification is most popularly used by connection-oriented, resource-controlling programs. An SNA Network Gateway is an example of such a program, since it controls access to an external computer network, and its clients are terminal emulation applications, which establish a communications session with a remote computer on the SNA network.

(For details on connection-oriented programs, see "Writing Request-Based System Services" in the *CTOS/Open Programming Practices and Standards manual.*) Briefly, a connection-oriented system service defines requests to

- Establish the connection with the client process and return a resource handle to that client. This is accomplished by defining an *open* type of request in the request definition file. The resource handle is used by the client in requests to manipulate the resource.
- Manipulate the resource allocated for the connection. These requests perform the actual work of the system service, such as reading from or writing to a file.
- Terminate the connection. This type of request is called a *close* request. It asks the system service to release any resources associated with the given handle and to terminate the connection with the client owning the handle.

The resource handle returned is a globally unique number associating the client with a specific instance of the system service, if multiple instances exist. The mechanism behind unique handles is very similar to the one used to create unique network handles. (For details, see "Routing by Handle" in the *CTOS Operating System Concepts Manual* in the section entitled "Interprocessor Communication.")

Example 8-1 shows a sample *Request.txt* file for a connection-oriented system service that uses routing by device specification.

The sample system service described in *CTOS / Open Programming Practices and Standards* runs unaltered on an XE-530 or an EISA/ISA-bus workstation if you use the *Request.txt* file shown in Example 8-1. However, you need to change its client programs.

Example 8-1 does not show how to deinstall an instance of the system service on a specified processor. You can define this deinstallation in either of two ways. (See "Deinstalling an Instance of a System Service," for details.)

Example 8-2 shows a client program that can access the system service on any processor. The user simply specifies the processor name (for example, *"[GP00]"*) as a runtime parameter to the client program.

Finally, Example 8-3 shows two enhancements you can make to the sample system service in *CTOS / Open Programming Practices and Standards*. These changes cause the system service to identify the name of the processor on which it is running and to return that name as part of the message to its clients.

Example 8-1. Sample Request.txt File Using Device Routing

```
:WsAbortRq:       0EF05h
:TerminationRq:   0EF05h
:SwappingRq:      0EF04h
:ChgUserNumRq:    0EF06h

:RequestCode:     0EF00h
:RequestName:     GetFooText
:Version:         4
:LclSvcCode:      0000h
:ServiceExch:     exchInstalledMastr
:sCntInfo:        2
:nReqPbCb:        0
:nRespPbcb:       2
:Params:          w(12), p(14), w(18), p(20), c(2,24)
:NetRouting:      rFh
:SrpRouting:      rDevice
```

Example 8-1. Sample Request.txt File Using Device Routing (cont.)

```
:RequestCode:      0EF01h
:RequestName:      DeinstallFooServer
:Version:          2
:LclSvcCode:       0000h
:ServiceExch:      exchInstalledMastr
:sCntInfo:         0
:nReqPbCb:         0
:nRespPbcb:        0
:Params:           none
:NetRouting:       NoRouting
:SrpRouting:       rRemote

:RequestCode:      0EF02h
:RequestName:      OpenFooServer
:Version:          3
:LclSvcCode:       0000h
:ServiceExch:      exchInstalledMastr
:sCntInfo:         0
:nReqPbCb:         1
:nRespPbcb:        1
:Params:           p(12), w(16), p(18), c(2,22)c(2,28)
:NetRouting:       DevSpec,OpenFh,CloseAtTermination
:SrpRouting:       rDevice

:RequestCode:      0EF03h
:RequestName:      CloseFooServer
:Version:          3
:LclSvcCode:       0000h
:ServiceExch:      exchInstalledMastr
:sCntInfo:         2
:nReqPbCb:         0
:nRespPbcb:        0
:Params:           w(12)
:NetRouting:       CloseFh,rFh
:SrpRouting:       rDevice
```

Example 8-1. Sample Request.txt File Using Device Routing (cont.)

```
:RequestCode:     0EF04h
:RequestName:     SwapClientFooServer
:Version:         2
:LclSvcCode:      0000h
:ServiceExch:     exchInstalledMastr
:sCntInfo:        2
:nReqPbCb:        0
:nRespPbcb:       0
:Params:          none
:NetRouting:      noRouting
:SrpRouting:      rDevice

:RequestCode:     0EF05h
:RequestName:     TerminateClientFooServer
:Version:         2
:LclSvcCode:      0000h
:ServiceExch:     exchInstalledMastr
:sCntInfo:        2
:nReqPbCb:        0
:nRespPbcb:       0
:Params:          none
:NetRouting:      noRouting
:SrpRouting:      rDevice

:RequestCode:     0EF06h
:RequestName:     ChangeUserClientFooServer
:Version:         2
:LclSvcCode:      0000h
:ServiceExch:     exchInstalledMastr
:sCntInfo:        2
:nReqPbCb:        0
:nRespPbcb:       0
:Params:          none
:NetRouting:      noRouting
:SrpRouting:      rDevice
```

Example 8-2. Client Program that Supports Device Routing

```
#define CheckErc                @
#define Delay                   @
#define ErrorExit               @
#define OutputToVid0            @
#define ReadKbdDirect           @
#define Syslit

#include <Ctostypes.h>
#include <Ctos.Lib.h>
#include <string.h>
#include <stdlib.h>

extern ErcType _far _plm CloseFooServer(Word wHandle);
extern ErcType _far _plm GetFooText(Word wHandle,
                     Pointer pbDataRet, Word cbDataRet,
                     Pointer psDataRet);
extern ErcType _far _plm OpenFooServer (Pointer pbDevSpec,
                     Word  cbDevSpec, Pointer pHandleRet);
extern ErcType _far _plm DeinstallFooServer (
                     Pointer pbDevSpec,Word  cbDevSpec);

#define CHECKONLY          1
#define DELAYRATE          10
#define ERCNOCHARAVAIL     602

Word hConnect;
Word cbRet;

char bKey;
char rgbServerData[80];

char pMsgIntro [] ="Press any key to close the connection
                with the server.\n\n";
char *pbDevSpec = NULL;
Word cbDevSpec = 0;
```

Example 8-2. Client Program that Supports Device Routing (cont.)

```
void main (int argc, void *argv[])
      {
      /* get device spec from user param */
      if(argc > 1) {
            pbDevSpec = argv[1];
            cbDevSpec = strlen(pbDevSpec);
            }


      /* open my connection with the foo server */
      CheckErc( OpenFooServer(pbDevSpec, cbDevSpec,
                        &hConnect));

      /* open my connection with the foo server
      CheckErc( OpenFooServer (&hConnect)); */

      /* display the intro message */
      OutputToVid0 (pMsgIntro, strlen(pMsgIntro));

      /* loop checking for a keyboard character, any one */
      while (ReadKbdDirect (CHECKONLY, &bKey) ==
                                    ERCNOCHARAVAIL)
            {

            /* get a message from the server */
            CheckErc (GetFooText (hConnect, &rgbServerData,
                        sizeof(rgbServerData), &cbRet) );

            /* display the message */
            OutputToVid0 (&rgbServerData, cbRet);

            /* now wait awhile before continuing */
            CheckErc (Delay (DELAYRATE));
            }

      /* close my connection */
      CheckErc (CloseFooServer (hConnect));
      ErrorExit (0);
      }
```

Example 8-3. Changes to Illustrate Device Routing

```
/* new version of ProcessDataRequest returns board name
 */
void ProcessDataRequest (RqDataType *pRq)
   {
   int  i = 0;
   Word wHandle = pRq->wHandle;

   /* check for valid handle */
   if (pRq->RqHead.userNum != rgHandles[(wHandle)])
      {
      pRq->RqHead.ercRet = ERCINVALIDHANDLE;
      return;
      }
   /* calculate the maximum string size */
   if (pRq->cbDataRet < strlen(pMsgRet) +
       strlen(rgbDeviceId))
      i = pRq->cbDataRet;
   else
      i = strlen (pMsgRet) + strlen(rgbDeviceId);

   /* put the message and size in the user's buffer */
   *((char *) pRq->pbDataRet) = 0;
   strncpy (pRq->pbDataRet, pMsgRet, i);

   /* attach device spec */
   if (i-strlen(pMsgRet) > 0)
      strncat(pRq->pbDataRet, rgbDeviceId,
              i-strlen(pMsgRet));
   *((Word *)(pRq->psDataRet)) = i;

   /* return an error OK condition */
   pRq->RqHead.ercRet = 0;
   }
```

Example 8-3. Changes to Illustrate Device Routing (cont.)

```
 /*
*  The following code comes later, inside of main()
*  function for program, but before ConvertToSys.
*/

  /* see if I'm running on a master */
  /* get config block */
  CheckErc(GetPStructure(0x2C8, 0, &pMySysConfigRet));
  /* check offset 33 */
  if(pMySysConfigRet->ClusterConfiguration == 2)
     fMaster = TRUE;
  else
     fMaster = FALSE;
  /* see what my board ID is if I'm on an SRP */
  if(fMaster) {
     erc = GetProcInfo(&i, &i, &i, rgbDeviceId,
                       DEVICEIDMAX, &cbDeviceId);
     if(erc == ercOK)
        /* null terminate the string */
        rgbDeviceId[cbDeviceId] = 0;
     else
        /* null string */
        rgbDeviceId[0] = 0;
  }
```

Deinstalling an Instance of a System Service

To deinstall an instance of a multi-instance system service on a specified processor, you need to define a deinstallation request in *Request.txt*. You can define it in either of two ways.

First, you can use device-specification routing and you can specify the processor on which to deinstall the system service using the network routing directive, DevSpec, as shown below:

```
:RequestCode:     0EF01h
:RequestName:     DeinstallFooServer
:Version:         13
:LclSvcCode:      0000h
:ServiceExch:     exchInstalledMastr
:sCntInfo:        0
:nReqPbCb:        1
:nRespPbcb:       0
:Params:          p(12),w(16)
:NetRouting:      DevSpec
:SrpRouting:      rDevice
```

Example 8-4 is an example of how the routed deinstall request is made.

Example 8-4. Routed Deinstallation Request

```
extern ErcType _far _plm DeinstallFooServer (
     Pointer pbDevSpec,Word  cbDevSpec);
     .
     .
     .
     if ( argc > 2 )
     {
          if ( *(char *)argv[2] == 'D' ||
                    *(char *)argv[2] == 'd' )
               CheckErc (DeinstallFooServer (pbDevSpec,
                    cbDevSpec));
          else
               OutputToVid0 ("Invalid deinstall argument.",
                                        27);
          ErrorExit (0);
     }
     .
     .
     .
```

Alternately, you can define the deinstallation request, as shown below:

```
:RequestCode:      0EF01h
:RequestName:      DeinstallFooServer
:Version:          2
:LclSvcCode:       0000h
:ServiceExch:      exchInstalledMastr
:sCntInfo:         0
:nReqPbCb:         0
:nRespPbcb:        0
:Params:           none
:NetRouting:       NoRouting
:SrpRouting:       rLocal
```

The above deinstallation request uses the local routing type (rLocal) for interprocessor routing. (For details, see "Local Routing," below.) This request must be issued on the processor where the system service is located.

Remote Routing

Remote routing (rRemote) is a commonly used type of request routing. It allows one instance of a system service to serve client processes on all processors and all client workstations. The client processes need not know where the system service resides. The request is routed automatically to the correct processor by the operating system.

Although a single instance of a system service typically is all that is needed, you should consider the circumstances and the type of service you are writing. In cases where a service is widely used, you might choose to install several intances of it instead using device routing (rDevice). If, however, you know you will always run a single instance of a service on a multiprocessor system, it is recommended that you use remote routing.

Example 8-5 shows a sample *Request.txt* file for a system service that uses remote routing. The sample system service described in *CTOS/Open Programming Practices and Standards* runs unaltered on the XE-530 or an EISA/ISA-bus workstation if you use the *Request.txt* file shown in Example 8-5.

Example 8-5. Sample Request.txt File Using Remote Routing

```
:WsAbortRq:        0EF05h
:TerminationRq:    0EF05h
:SwappingRq:       0EF04h
:ChgUserNumRq:     0EF06h

:RequestCode:      0EF00h
:RequestName:      GetFooText
:Version:          3
:LclSvcCode:       0000h
:ServiceExch:      exchInstalledMastr
:sCntInfo:         2
:nReqPbCb:         0
:nRespPbcb:        2
:Params:           w(12), p(14), w(18), p(20), c(2,24)
:NetRouting:       rFh
:SrpRouting:       rRemote

:RequestCode:      0EF01h
:RequestName:      DeinstallFooServer
:Version:          2
:LclSvcCode:       0000h
:ServiceExch:      exchInstalledMastr
:sCntInfo:         0
:nReqPbCb:         0
:nRespPbcb:        0
:Params:           none
:NetRouting:       noRouting
:SrpRouting:       rRemote

:RequestCode:      0EF02h
:RequestName:      OpenFooServer
:Version:          2
:LclSvcCode:       0000h
:ServiceExch:      exchInstalledMastr
:sCntInfo:         0
:nReqPbCb:         0
:nRespPbcb:        1
:Params:           p(12), c(2,16)
:NetRouting:       OpenFh,CloseAtTermination
:SrpRouting:       rRemote
```

Example 8-5. Sample Request.txt File Using Remote Routing (cont.)

```
:RequestCode:      0EF03h
:RequestName:      CloseFooServer
:Version:          2
:LclSvcCode:       0000h
:ServiceExch:      exchInstalledMastr
:sCntInfo:         2
:nReqPbCb:         0
:nRespPbcb:        0
:Params:           w(12)
:NetRouting:       CloseFh,rFh
:SrpRouting:       rRemote

:RequestCode:      0EF04h
:RequestName:      SwapClientFooServer
:Version:          2
:LclSvcCode:       0000h
:ServiceExch:      exchInstalledMastr
:sCntInfo:         2
:nReqPbCb:         0
:nRespPbcb:        0
:Params:           none
:NetRouting:       noRouting
:SrpRouting:       rRemote

:RequestCode:      0EF05h
:RequestName:      TerminateClientFooServer
:Version:          2
:LclSvcCode:       0000h
:ServiceExch:      exchInstalledMastr
:sCntInfo:         2
:nReqPbCb:         0
:nRespPbcb:        0
:Params:           none
:NetRouting:       noRouting
:SrpRouting:       rRemote
```

Example 8-5. Sample Request.txt File Using Remote Routing (cont.)

```
:RequestCode:       0EF06h
:RequestName:       ChangeUserClientFooServer
:Version:           2
:LclSvcCode:        0000h
:ServiceExch:       exchInstalledMastr
:sCntInfo:          2
:nReqPbCb:          0
:nRespPbcb:         0
:Params:            none
:NetRouting:        noRouting
:SrpRouting:        rRemote
```

Local Routing

Local routing (rLocal) is the simplest form of request routing on a multiprocessor system. In general, this type of routing is used when a request should be served on the processor where it was generated.

When a system service uses local routing for its requests, it must be installed on the same processor as its clients. If its client (processes) are running on client workstations, the service must be installed on the processor to which those client workstations are physically connected.

If a system service uses local routing, multiple instances of it can be installed on a multiprocessor system, but only local clients can communicate with each one. In other words, if a user installs the system service on each of two GP processors of a shared resource processor system, users on each GP can talk to their local copy of the system service. However, users on each GP cannot communicate with the copy of the service on the other GP. Likewise, if the system service is installed on an FP, only other programs running on that FP can communicate with it.

Because of the above restrictions, use local routing sparingly.

Other Routing Methods

The other interprocessor request routing methods are rarely used, except by the operating system itself. However, their use is not discouraged. If your system service needs to use one of them, it should do so.

Bear in mind, however, that some of the rarely used routing types require specially formatted connection handles. These requirements are described below. (See "Use of Handles on Multiprocessor Systems.")

Enhancing Remote Data Transfer Requests on SRPs

If you are writing a system service (device driver) that controls a peripheral device such as a disk or scanner and your driver is required to move large amounts of data rapidly and efficiently, you can increase data movement using the direct memory address (DMA) hardware. (For a general discussion of DMA and device drivers, see "Bus Address Management" in the *CTOS Operating System Concepts Manual.*)

DMA transfers performed between the local processor memory and the device are supported on all multiprocessor systems. On shared resource processors, however, your device driver can make DMA transfers between processors as well. If the peripheral device is located on a remote processor, you can use remote DMA to transfer the data directly from the device to the client's memory on another processor or vice versa. Using remote DMA in a read request, for example, eliminates the extra system overhead involved in copying data to be read into an inter-CPU communication (ICC) buffer and then copying it again from the ICC buffer to the requesting client's memory.

To use remote DMA, you need to design the request block such that the first pb/cb pair describes the read/write data buffer.

The Read and Write requests are examples of requests formatted to meet the requirements of remote DMA. (For details, see the request block formats for these requests in the *CTOS Procedural Interface Reference Manual.*)

In *Request.txt*, the Read request would appear as shown below:

```
:RequestCode:     0023h
:RequestName:     Read
:Version:         0
:LclSvcCode:      0000h
:ServiceExch:     exchFileSys
:sCntInfo:        6
:nReqPbCb:        0
:nRespPbcb:       2
:Params:          fh, pbcb0, lfa, p(24), c(2,28)
:NetRouting:      rFh, RW
:SrpRouting:      rHandle, dmaRemote
```

Use of Handles on Multiprocessor Systems

On multiprocessor systems, most request routing types use the same format for handles as is used on single-processor systems. Because almost all system services use these routing types, system services for multiprocessor systems rarely need to use special handles.

The following interprocessor routing types use standard connection handles:

rLocal
rRemote
rDevice
rBroadcast

The following routing types require special handles:

rHandle
rFileId
rLineNumber

The formats of the connection handles are described below.

The Standard Connection Handle

Most interprocessor routing types use standard connection handles. These handles have the following format:

Bit 15: reserved for network use.

Bits 13 and 14: set to 1s if system service is installed on a server.

Bit 12: Reserved. Should be set to 0.

Bits 11 to 0: Available for use by the system service.

Note, however, that the system service and its client should not attach meaning to any bits within the handle. The service cannot pass information to the client by returning a particular value in the connection handle. CTOS guarantees that both the service and the client will perceive a unique handle value, but *that value may not be the same* for both the client and the service.

Non-Standard Handle Types

Requests that are routed rHandle, rFileId, or rLineNumber require specially formatted handles. The format for each type is described below.

rHandle: Connection handles for requests that are routed rHandle should have the following format:

Bits	Description
13 through 15	Standard.
10,11,12	FPIndex. See description below.
0 through 9	Available for use by service.

The FPIndex bits contain the index of the file processor (shared resource processor FP or GP/SI, or EISA-bus multiprocessor workstation EP) on which the service is installed. In other words, on a shared resource processor, if the service is installed on FP00, the FPIndex is 0; if the service is installed on FP01, the FPIndex is 1. On an EISA/ISA-bus multiprocessor workstation, the FPIndex is 0 if the service is installed on an EP00 and 1 on an EP01, and so forth.

rFileId: Connection handles for requests that are routed rFileId should have the following format:

Bits	Description
12 through 15	Standard.
8 through 11	Available for use by the service (4 bits).
0 through 7	Processor Slot ID (for example, 77h)

rLineNumber: Connection handles for requests that are routed rLineNumber should have the following format:

Bits	Description
12 through 15	Standard.
8 through 11	Available for use by the service (4 bits).
0 through 7	Cluster Line Number. See description below.

The cluster line number is a unique cluster-line index for the entire SRP starting at cluster line 1. For example, CP00 (or GP00) owns lines 1 and 2, CP01 (or GP01) owns lines 3 and 4, and so on.

On an EISA-bus multiprocessor workstation, the first Cluster IOP owns cluster line number 1, the second owns cluster line 2, and so forth.

Section 9
The Synchronous CommLine Interface

Introduction

This section describes extensions to the CommLine interface for synchronous data communications.

The CommLine Interface

The CommLine interface is discussed in detail in "Communications Programming" in *CTOS / Open Programming Practices and Standards.* That section explains the components of the interface and gives general guidelines for using it. The section also gives general examples of initializing the serial controller and includes a sample communications interrupt service routine. Unless you already understand the CommLine interface, read that section before continuing.

To recap the information in *CTOS / Open Programming Practices and Standards*, the CommLine interface is a fairly low-level interface used for synchronous data communications. The interface is designed to provide as much device-independence as possible without sacrificing performance. The CommLine interface allows a program that uses it to operate on any workstation, using any serial port.

However, because performance was the primary consideration in its design, the CommLine interface requires the programmer to perform a certain amount of serial controller initialization via direct register writes.

Extensions to the Traditional CommLine Interface

The traditional CommLine interface offered limited choices. A program called InitCommLine and received two port addresses: a data port and a control port. The program used the data port to exchange words of user data with the serial controller. The program used the control port to perform any initialization or control that required direct reads or writes to the registers on the serial controller.

Additional operations, ReadCommLineStatus and WriteCommLineStatus, controlled RS-232 signals that were not controlled directly by 8274-type serial controllers. This allowed the application to query and set these signals in a device-independent fashion.

Lastly, the ChangeCommLineBaudRate operation allowed the application to set the speed of data transfer in a device-independent way.

All these calls remain in the current version of the interface, but extensions have been added. The subsections below address each operation in turn.

InitCommLine

InitCommLine has undergone substantial changes from its traditional form for CTOS/XE 3.0 and later versions of the operating system. However, these changes have been made in such a way that it remains compatible with existing programs. The mechanism by which this is accomplished is as follows.

When a new feature is added to the available hardware, a new field is added to the Communications Line Configuration Block and to the Communications Line Return Area for subsequent versions of the operating system.

Programs that know of and desire to use such a feature can set that field in the Communications Line Configuration Block to some specific nonzero value. If the desired feature is present in the current hardware, the operating system sets the corresponding field in the Communications Line Return Area to a nonzero value.

This way, the program knows that the desired feature is present in the hardware and supported by the operating system. If the feature is not available, the operating system returns zero in the appropriate field of the Communications Line Return Area. The program can then continue or terminate, as appropriate for its application.

The operating system is also able to identify the features that are not known to a program. This is because the program specifies the size of the Communications Line Configuration Block and of the CommLine Return Area in its call to InitCommLine. Any fields beyond the specified size are unknown to the program and, therefore, clearly not used by it.

Differences Between 8274 and 8530 Communication Controllers

The 8274-type and 8530-type serial communication controllers are similar but not quite register-compatible. This has at least some effect on almost all synchronous communications programs. The table below shows which registers differ between the two controllers.

Table 9-1. 8274 and 8530 Register Differences

Register Number and Bit Position*	8274	8530
RR0, bit 1	Interrupt pending	Zero count
WR0, bits 3-5		
= 001	Send abort	Point high (access regs 9-15)
= 011	Channel reset	Send abort
WR1		
bit 2	Status affects vector	Parity is special condition
bits 3-4		
= 00	RxInt/DMA disable	RxInt disable
= 10	Int. on Rx,special, and parity	Int. on Rx and special
= 11	Int. on Rx and special	Int. on special only
bit 6	Always 0	Ready/DMA reg.function
WR2 (only channel A)	Interrupt control functions	Channel A interrupt vector
RR3	N/A	Similar to interrupt pending
WR9, bit 0	N/A	Status affects vector
bits 1-5	N/A	Interrupt control functions
bits 6-7	N/A	Channel reset

* Bits are numbered 76543210

ChangeCommLineBaudRate

ChangeCommLineBaudRate allows a program to modify a synchronous channel's data rate dynamically.

8274-type serial controllers are not affected by this operation, though errors may occur if a data transfer is in progress when this operation is called.

If the serial controller is an 8530-type controller, this operation sets it to the new data rate and, if specified, changes the clock source to external.

Lastly, on channels that do not support separate transmit and receive baud rates, the value of the *iRxTx* parameter to the ChangeCommLineBaudRate operation must be 0 (both). If a program attempts to set the receive and transmit baud rates separately on hardware that does not support that feature, status code (7 or 61) is returned, and the operation is not performed.

ReadCommLineStatus

ReadCommLineStatus performs the same functions as before, but has additional case values. ReadCommLineStatus can now read the following signals:

Secondary receive data (SRxD)
Data set ready (DSR)
Ring indicator (RI)
Secondary carrier detect (SCD)
Secondary clear-to-send (SCTS)
Signal quality

Note that not all hardware supports the SCD, SCTS and Signal Quality signals. See your hardware documentation for more information.

WriteCommLineStatus

Like ReadCommLineStatus, WriteCommLineStatus is essentially unchanged but supports additional case values. WriteCommLineStatus can now set or clear the following signals:

Secondary transmit data (SRxD)
Secondary request to send (SRTS)
Rate select
Select standby
Data terminal ready (DTR)

Programs that use DMA *must* use WriteCommLineStatus to set or clear the DTR signal. Programs that do not use DMA must do so using direct serial controller port I/O.

Using DMA with Synchronous Data Communication

CTOS now provides a device-independent interface for programs that want to use DMA for communications I/O. DMA for serial communications is supported only on certain hardware, such as the B39 workstation and the GP processor on the XE-530. (See your hardware documentation for more specific information.) Programs can transfer blocks of data as large as 64K bytes using DMA, though a smaller block size is more commonly used.

This section describes how to use the CommLine DMA interface, which includes the following system-common procedures:

TransmitCommLineDma (alias: XmitCommLineDMA)
ReceiveCommLineDma (alias: RecvCommLineDMA)
GetCommLineDMAStatus

Note that because the CommLine DMA operations are system common procedures, they can be called directly from interrupt service routines.

Initializing Communications DMA

If a communications program wants to use DMA, it must specify that in its call to InitCommLine.

The Communications Line Configuration Block contains a field, *fDMA*, which allows the program to indicate that it wants to use DMA for communications I/O. To request DMA, the program must set the value of this field to 0FFh (255).

After it calls InitCommLine, the program must check the value of the *DMAavailable* field of the Communications Line Return Area. If the value is 0FFh (255), DMA hardware is available for the program to use. If the value is 0, no DMA hardware is available.

If communications DMA hardware is present, InitCommLine sets the serial controller to use DMA I/O mode instead of interrupt I/O mode. This requires setting certain bits in write register 1 (bits D7, D6, D5) and write register 14 (bit D2) of the serial controller. If a DMA-using program writes to either of these registers, it must be careful to preserve the settings of the DMA mode control bits.

In addition, if a DMA-using program needs to assert or deassert the RS-232 DTR signal, it must use the WriteCommLineStatus procedure to do so.

Finally, note that when DMA is requested, InitCommLine performs its normal functions plus setting up the serial controller to use DMA, but nothing more. The client communications program is still responsible for completing the initialization of the serial controller.

TransmitCommLineDMA

The TransmitCommLineDMA procedure (called XmitCommLineDMA in BTOS 3.x) sets up the DMA controller to transfer data directly from memory to the serial communication controller. This procedure returns immediately, since it merely sets up a future data transfer. It does not wait for the data transfer to occur.

The TransmitCommLineDMA procedure takes three parameters: the CommLine connection handle, a pointer to a data buffer, and the size of the buffer. After TransmitCommLineDma returns to its caller, the DMA controller transfers this data one byte at a time from the specified buffer as the serial controller requests it. The transfers continue until the entire buffer has been sent. For efficient use of the system data bus, the buffer should be either word-aligned or double word-aligned.

When the specified number of bytes have been sent, the serial controller sends the CRC check value and then generates an External/Status interrupt. The program's External/Status ISR should verify that the correct number of bytes was transmitted and then call TransmitCommLineDma with the next buffer of data. The serial controller resumes DMA transfers before the TransmitCommLineDma call returns.

ReceiveCommLineDMA

The ReceiveCommLineDMA procedure (called RecvCommLineDMA in BTOS 3.x) sets up the DMA controller to transfer data directly from the serial communication controller to memory. It is the reverse of the TransmitCommLineDma procedure. Like the transmit procedure, ReceiveCommLineDma returns immediately, since it merely sets up a future data transfer. It does not wait for the data transfer to occur.

The ReceiveCommLineDma procedure takes four parameters: the CommLine connection handle, a pointer to a data buffer, the size of the buffer, and a pointer to a returned count of bytes transferred. After ReceiveCommLineDma returns to its caller, the DMA controller transfers data one byte at a time to the specified memory buffer, as the serial controller receives it. The transfers continue either until a complete data frame is received or until the transfer count reaches the size of the buffer. As with transmit buffers, the receive buffer should be word-aligned or double word-aligned.

When the serial controller detects the end of a data frame (EOF) or a full buffer, it generates a Receive Special interrupt. The program's receive ISR should verify that a complete frame was received successfully then call ReceiveCommLineDma with the next available buffer for data.

Note that the 8530 controller does not indicate EOF until after all characters have been read from its buffer. If the DMA receive buffer is too small, EOF may be detected but never reported. For this reason, it is important to ensure that the receive buffer is slightly larger than the largest possible incoming frame.

The ReceiveCommLineDma procedure also returns the number of bytes successfully received by the previous buffer. This allows the program to know how much valid data is present in the buffer without making any additional procedure calls.

The serial controller resumes DMA transfers before the ReceiveCommLineDma call returns.

GetCommLineDMAStatus

The GetCommLineDMAStatus procedure returns the count of bytes transferred for the current DMA operation in both the transmit and the receive direction.

If called from the External Status ISR, it allows the program to determine whether a transmission was completed successfully. To do so, the program simply compares the returned byte count to the size of the transmit buffer. If they are equal, all bytes were sent successfully.

If called from the Receive Special ISR, it allows the program to determine the size of the data in the receive buffer. While the ReceiveCommLineDma procedure also provides this information, some programs may need to use GetCommLineDMAStatus anyway. For example, if the program needs to verify the received data in some way before requesting the next transfer, it should determine the size of the received data by calling GetCommLineDMAStatus.

Implementing X.21 (1984) Protocol Support

Certain hardware models, namely the XE-530 GP/CI processor set, include special hardware that supports the X.21 signalling protocol. On some other systems, X.21 can be used if you install convertor hardware. This section describes the interface to that hardware and the programming tasks required to use it.

What Is the X.21 Protocol?

X.21 is a digital signaling protocol defined by the CCITT (Consultative Committee for International Telegraph and Telephone). It allows a DTE (Data Terminal Equipment) to set up and clear data calls by exchanging signals with the DCE (Data Circuit-Terminating Equipment) connected to it. The X.21 protocol uses a combination of data and control signals to accomplish this task. This section summarizes some of the main features of the X.21 standard. For more detailed information on X.21, see the CCITT X.21 standards document.

In the X.21 protocol the DTE controls two signal lines, Transmit (T) and Control (C). The DTE passes data and signaling information to the DCE over the Transmit line, and passes control information to the DCE on the Control line. The T and C lines are similar in function to the RS-232 TxD and RTS signals, though the T and C lines perform additional duties.

The DCE also controls two signal lines, Receive (R) and Indication (I). The DCE passes user data and signaling information to the DTE on the Receive line, and passes control information to the DTE on the Indication line. The R and I lines are similar in function to the RS-232 RxD and CTS signals, though the R and I lines perform additional duties.

The primary additional duties performed by the T and R lines are the sending and receiving of inband digital signaling information between the DCE and DTE. This signaling information is used in a way that is analogous to the call setup tones you hear when placing a telephone call.

When the line is idle, the DCE always indicates "ready," the equivalent of dial tone. When the DTE initiates a call, the DCE returns a digital "dial tone" indicating that the DTE can continue. The DTE and DCE then exchange a series of call-progress messages while they set up a connection with a remote computer. Finally, the DTE and DCE enter a data-transfer mode for the duration of the call.

Features of the X.21 Support Hardware

The serial controller performs most X.21 support functions, but there are a few that require special hardware. When the serial controller is in its X.21 mode, it recognizes only data frames that are preceded by two X.21 SYNC characters. This allows it to recognize any valid X.21 data frame, but prevents it from recognizing certain special signaling patterns that can appear in the incoming data bitstream.

Dedicated hardware has been added to detect these special bit patterns. This hardware scans the incoming bitstream for one of four patterns that indicate special conditions in the 1984 X.21 protocol. When it detects one of the patterns, it first verifies that the Indication line is in the correct state for the pattern to be valid. The X.21 hardware then writes a value in a special register and changes the state of the CTS signal to the serial controller.

This causes the serial controller to generate an interrupt whenever the X.21 hardware detects one of the special bit patterns. Note that the X.21 hardware detects only incoming patterns; it does not generate outgoing bit patterns. That is the job of the application program.

Three of these bit patterns must be held by the DCE for at least 24 consecutive bits. They are detected by the X.21 hardware when two consecutive bytes (16 bits) contain the pattern. The fourth pattern must be held by the DCE for 16 bits and must appear in the data bitstream only once, immediately after the Indication line goes "on." This pattern indicates successful call completion.

See Table 9-2 for a description of each bit pattern and its required Indication line state.

Table 9-2. X.21 Status Indication Bit Patterns

Bit Pattern	I-Line State	Meaning
00000000 00000000 00000000	Off	DCE Not Ready
11111111 11111111 11111111	Off	DCE Ready
01010101 01010101 01010101	Off	DCE Controlled, not Ready
11111111 11111111	On	Ready For Data (Call Setup Complete)

When the serial controller detects a change in the state of CTS, it generates an External/Status interrupt. The communication program's External/Status ISR must then check that the cause of the interrupt is the X.21 hardware (that the state of CTS has changed) and then read the X.21 hardware register. InitCommLine returns the port address of the X.21 hardware register in the *ioX21* field of the Communications Line Return Area.

The possible values of bits 0-2 of the *ioX21* port are as follows. The decimal values are listed with the corresponding binary values in parentheses.

Value	Description
1 (001b)	DCE Not Ready
2 (010b)	DCE Ready
3 (011b)	DCE Controlled, not Ready
4 (100b)	DCE Ready for Data

The X.21 hardware also has the ability to store one bit pattern in a temporary buffer. Therefore, if a second bit pattern is detected while the first is being processed by the program's ISR, the X.21 hardware stores that pattern until the ISR finishes. It then changes the state of CTS again to generate another interrupt. Note that even if the X.21 hardware has a pattern stored in its buffer, it continues to check the incoming bitstream. If it detects a new pattern, it overwrites the one that is currently stored.

Initializing a Communications Line with X.21 Support Enabled

To request use of the X.21 hardware, a communications program must set the *fX21* flag in the Communications Line Configuration Block to the value 0AAh. This value requests full X.21 support.

If the X.21 hardware is present, InitCommLine returns the port address of the X.21 hardware in the *ioX21* field of the Communications Line Return Block. If the hardware is not present, InitCommLine returns a nonzero status code.

After a successful call to InitCommLine, the X.21 hardware for the channel is inactive. To activate the X.21 hardware, the program must write a one to bit zero of the *ioX21* port. After that happens, the X.21 hardware begins scanning the input stream for its special bit patterns.

The program can disable the X.21 hardware at any time, by writing a zero to bit zero of the X.21 hardware port. The program can then reenable it later by writing a one. The program should be careful to write only to bit zero of the X.21 hardware port, however. Other bits are used internally by the Comm Nub.

Note also that programs that use the X.21 hardware must initialize the serial controller to enable the External/Status interrupt. Otherwise, changes in the CTS signal by the X.21 hardware do not cause an interrupt.

Using the X.21 Support Hardware in Drivers-Only Mode

Using the X.21 support hardware in drivers-only mode disables the pattern-recognition circuitry. This allows programs to use the 1981 X.21 protocol, which does not include the bit patterns described above.

To request use of X.21 in drivers-only mode, the communications program must set the *fX21* flag in the Communications Line Configuration Block to the value 0FFh.

If X.21 is supported by the hardware, InitCommLine returns status code 0. However, it does not return a port address in the *ioX21* field of the Communications Line Return Area because the pattern detection hardware is not used in this mode. If X.21 is not supported by the hardware, InitCommLine returns a nonzero status code.

Using V.35 Support Hardware

As with X.21, only certain hardware supports the V.35 interface. However, using this interface is almost a nonevent for a communications program. V.35 is an electrical interface and appears to the software to be the same as RS-232 to communications programs, except that it supports faster data rates.

Existing programs need no modification to use the V.35 support hardware. When InitCommLine receives a Communications Line Configuration Block that is too small to contain the *fV35Mode* flag, it simply uses the current jumper settings for the requested communications port. So, if an existing program attempts to use a port that is jumpered for V.35, the attempt succeeds. Note, however, that such programs still support only the data rates they supported on RS-232. They may or may not work at higher V.35 rates.

New programs should set the *fV35Mode* flag in the Communications Line Configuration Block. This flag informs InitCommLine that the program wants to use V.35 mode. If the requested communications port supports (and is jumpered for) V.35, InitCommLine sets the *fV35Avail* field in the Communications Line Return Area to TRUE. If the requested communications port does not support (or is not jumpered for) V.35, InitCommLine sets the *fV35Avail* field to FALSE, and returns a nonzero status code.

Programs that explicitly check for V.35 support, as described above can use the higher data rates supported by the V.35 interface. Those that do not verify that V.35 hardware is present should restrict themselves to data rates supported by both RS-232 and V.35.

Section 10
Creating a Dynamic Link Library

Introduction

This section describes how to create a dynamic link library (DLL). A DLL is a collection of object module procedures stored in a disk file, which may be loaded into memory when called by client applications. DLLs are supported on CTOS III and higher operating systems.

In this section, we address all aspects of creating a DLL, from the tools you need to build it, to how a client application actually uses it. In between, we illustrate some of the problem areas programmers have encountered in writing DLLs on their own.

Assumptions

We recommend you use CTOS Microsoft C to create DLLs. For this reason, we present an example DLL program coded in CTOS Microsoft C and compiled using CTOS Microsoft C compiler options. We assume you, as the DLL writer, are familiar with writing CTOS Microsoft C applications to be run on CTOS III operating systems. In addition, we assume you have some familiarity with DLL concepts and terminology as used in the following supporting documentation.

Supporting Documentation

This section does not provide all the details on how to build applications or how to call operating system or language procedures. Each of the manuals below contains detailed information on DLLs:

- *CTOS Microsoft C Programming and Installation Reference Manual*
- *CTOS Operating System Concepts Manual*
- *CTOS Programming Utilities Reference Manual: Building Applications*
- *CTOS Procedural Interface Reference Manual*

In this section, we refer you to the appropriate manual for background information on subjects not fully covered here. (For a description of each of these manuals, see "Related Documentation" in "About This Manual.")

Note: *The DLL example uses loadtime dynamic linking. A DLL also can be dynamically linked at runtime. For details on runtime dynamic linking, see "Dynamic Link Libraries," in the* CTOS Operating System Concepts Manual.

Software and Tools

This section assumes you have the appropriate files and software available to create a DLL. As previously mentioned, you need to have CTOS III and the CTOS Microsoft C Compiler software installed on your system. In addition, you should have installed the Standard Software 12.3 Development Utilities. The files you need are described below.

The following files are packaged with Standard Software 12.3 Basic System Utilities:

File	Description
System.dll	Is a DLL file that allows your DLL to make calls to the CTOS Microsoft C library.
CtosLib.dll	Is a DLL file that mediates object module procedures in *Ctos.lib*. It is also used by the file *System.dll*.

The following files are packaged with Standard Software 12.3 Development Utilities:

File	Description
CtosLibimp.lib	Is the DLL imports library, which defines the contents of *CtosLib.dll*.
Systemimp.lib	Is the DLL imports library, which defines the contents of *System.dll*.

The following files are packaged with CTOS Microsoft C:

File	Description
MC0_Dll.obj	Is an object module you must link with your DLL to arrange segments in a default order (for DLLs written in CTOS Microsoft C).
DLL_Term.obj	Is an object module you can link with your DLL to perform C runtime cleanup.
LLibCDll.lib	Is a C library you must link with your DLL to make C library calls. Our example lists this library in the Linker configuration file, *LinkerConfig.sys*. (See Example 10-8.)

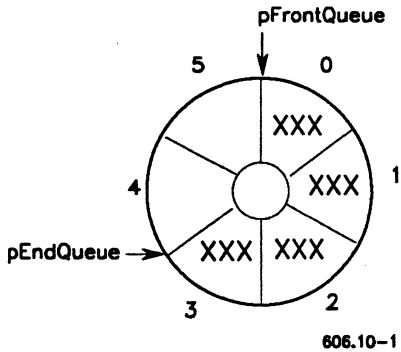
If you plan to use the CTOS Microsoft C **NMake** utility to build your DLL (or any of its clients), you must have installed the Task Management Service and the Environment Service packaged with CTOS Microsoft C.

Introducing the DLL Example

The DLL example implements a global queue that can be accessed by multiple clients. Its purpose is to illustrate the concepts associated with developing a DLL on CTOS.

See Figure 10-1. The queue is a circular FIFO list. Applications can add entries to the end and delete entries from the front. The queue is persistent. While it is not being used by any client, its contents are saved to a disk file.

Figure 10-1. Queue DLL



The first client to call a DLL procedure causes the queue to be read into memory. The queue remains in memory until the last client to use it terminates. This causes the queue to be written to a disk file.

Perhaps more important than what the example does is that it illustrates some of the more troublesome areas of writing DLLs. Specifically it shows how to handle the following:

- Mediating medium-model library procedures
- Setting up an initialization procedure
- Establishing termination procedure(s)
- Using a semaphore to handle global data
- Setting up instance and global data variables

The example calls C library routines. There is really nothing unusual about this in the code but you should be aware that doing so requires that you specify certain CTOS Microsoft C compiler options that aren't discussed in detail in this section. Although the file that builds the DLL (*Dll.mak*) includes these options, we leave their description to the CTOS Microsoft C documentation.

With any code you write, it is important to perform error checking. In the example, error checking follows a pattern of cleaning up (clearing and closing semaphores, and so forth) and exiting. Although several lines of code are dedicated to this purpose, we intentionally focus comments on DLL issues, and assume you handle error checking on your own.

DLL Source Files

The DLL consists of the source files listed below:

File Name	Contents	Shown in
Queue.c	C source code	Example 10-1
Queue.def	Module definition file	Example 10-2
Dll.mak	DLL makefile	Example 10-3

These files are shown in the listings at the end of this section. To run this program, we recommend you use the electronic versions of the files on the diskettes packaged with the Standard Software Development Utilities, rather than typing in the contents of the listings shown in this document.

Although the DLL example is a simple program, it seems long if you're just browsing through the hard copy of these files. To avoid interrupting the text flow with pages of source code, we present the source-file listings in their entirety at the end of this section.

Overview of Queue.c

Before focusing on the DLL issues, we provide an overview of the source file, *Queue.c*, shown in Example 10-1. To do this, we take a straightforward approach by scanning its contents from beginning to end. If you are already very familiar with CTOS Microsoft C and creating queues, this subsection may be elementary to you. Simply skip over to the subsection entitled "Setting Up the Initialization Procedure."

Queue.c Declarations

As with most source files, this one starts with declarations and data type definitions. In fact, many of the type definitions this example uses are common to most CTOS applications. Ordinarily you would define these in separate include files. We simply wanted to show you which ones the example uses. If you're just getting acquainted with CTOS, see "Naming Conventions" at the beginning of this manual, for details on data types and CTOS naming conventions.

Global Data Declarations

This takes us to the global data declarations. The example defines the default data type as nonshared (or instance) data. However, it also uses global (or shared) data. To override the default, the global data is declared in the declarations and defined using a Segments statement in the module definition file, *Queue.def*. (For details, see "Setting Up Global Data Variables.")

The queue is a global structure. As such, it needs to be available to any application that wants to use it. The queue pointer variables *pFrontQueue*, *pEndQueue*, and *PQueueMem* must be shared and known to all users. Furthermore, because the user count *nUsers* is global, the DLL can detect when the queue is not being used and can save it to disk before the operating system removes the DLL from memory.

Instance Data Declarations

Following the global declarations, we see a couple of instance-data variables, *nAdds* and *nDeletes*. These are defined simply to keep track of the number of times each client adds or deletes a queue entry.

Queue Pointer Procedures

The first procedures we encounter in *Queue.c* are short internal ones to manipulate queue pointers. As you recall, the queue is implemented as a circular array. To understand how this DLL example works, you should be aware of a few issues with circular queues. We describe these below.

IncrementQueuePtr

IncrementQueuePtr increments the offsets of the global queue pointer variables *pFrontQueue* and *pEndQueue* one byte at a time. Queue pointer offset values can be incremented until they reach the queue's maximum size (QUEUESIZ-1). At QUEUESIZ-1, IncrementQueuePtr resets the offset to *pQueueMem*, which is the address of the actual start of the queue. This essentially creates a circular queue.

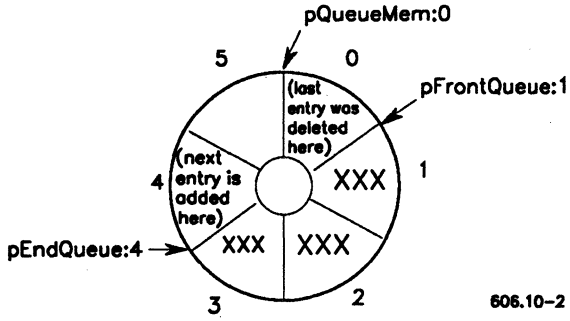
To illustrate this, let's assume a simpler queue than our example. Although the example queue is actually declared as 8000h bytes long (a constant value in the declarations) and can have variable-sized entries up to 0FFh bytes each, let's suppose the queue is 6 bytes long with 1-byte long entries. We will also assume that the queue starts at offset 0 (*pQueueMem*:0).

When this queue is read into memory for the first time, the global pointer variables *pFrontQueue* and *pQueueMem* reference the beginning of the queue and have an offset of 0.

As clients add and delete queue entries, the offsets of these pointer variables change. *pFrontQueue* always points to the first queue entry and *pEndQueue*, to the first free byte (location where the next queue entry should be written).

With each queue entry added, IncrementQueuePtr increments the offset of *pEndQueue* by the size of the entry (in this simple case, 1 byte) to the next free byte at the end of the queue. With each queue deletion, it increments the *pFrontQueue* offset to reference the first byte of the next entry. (See Figure 10-2.)

Figure 10-2. IncrementQueuePtr



When incrementing *pFrontQueue* or *pEndQueue* causes the offset values to exceed the queue size (that is, when the offset equals 5; remember that it's 0 based), IncrementQueuePtr resets the offset to be that of *pQueueMem* (in this case, 0). This "contains" the queue in the 6-byte queue memory area referenced by *pQueueMem*.

If pointers weren't reset, *Queue.c* would have to handle ever increasing values. The circular queue simplifies pointer calculations.

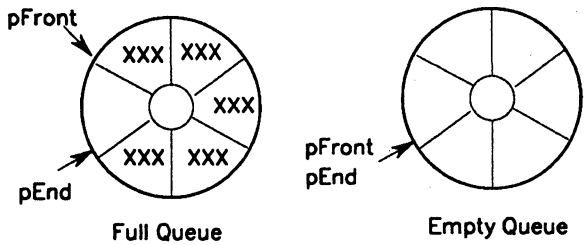
QueueUsedSpace

QueueUsedSpace calculates the amount of space remaining in the queue to determine if, for example, it can add a new queue entry of a specified size.

QueueFreeSpace

To determine the amount of free space, we must be able to tell a full queue from an empty one. The queue pointer procedures always leave at least one free byte to make this distinction. (See Figure 10-3.)

Figure 10-3. Leaving-One-Byte-Free



ReadQueueIntoMemory

ReadQueueIntoMemory reads the queue into memory at first use by the first client. This reading procedure is really a subroutine of the initialization procedure, QueueInitialization. (We discuss QueueInitialization in detail below. See "Setting up the Initialization Procedure.")

WriteQueueFromMemory

WriteQueueFromMemory writes the queue from memory to a disk file.

WriteQueueFromMemory is really a termination subroutine and is only called at termination of the last client. (We discuss queue termination in detail below. See "Using a Termination Procedure.")

Initialization and Termination Procedures

Moving along, we encounter the respective initialization and termination procedures QueueInitialization and QueueTermination.

QueueInitialization calls ReadQueueIntoMemory upon the detection of the first user. This happens when the global variable *nUsers* is 0.

QueueTermination calls WriteQueueFromMemory to write out the contents of the queue when *Queue.c* detects that the last queue user is terminating. This happens when decrementing the global variable *nUsers* results in a value of 0.

Export Procedures

Finally we arrive at the three export procedures, QueueAdd, QueueDelete, and QueueStatus.

Generally, QueueAdd and QueueDelete manipulate global queue pointers as clients add and delete entries from the queue. QueueStatus reports the number of deletions and additions to the queue on a per user basis. It can do this because *nAdds* and *nDeletes* are nonshared variables.

Mediating Medium-Model Library Procedures

The CTOS semaphore management operations declared at the beginning of *Queue.c* are system operations (that is, requests or system-common procedures). These operations do not require any special handling in a DLL. If, however, you do include library procedures from any of the medium model CTOS standard operating system libraries (*Ctos.lib*, *CtosToolKit.lib*, or *Enls.lib*) or from any external library not compiled in large or huge model, you need to mediate the procedures so that they can be called from a DLL.

The DLL, *CtosLib.dll*, and the import library, *CtosLibimp.lib*, work together to mediate many (but not all) of the object module procedures in the CTOS operating system libraries *Ctos.lib*, *CtosToolkit.lib*, and *Enls.lib*. If your DLL uses a CTOS object module procedure that is not resolved by *CtosLibimp.lib*, you need to create another mediated DLL that contains those procedures. (For details on how to do this, see Appendix D in the *CTOS Programming Utilities Reference Manual: Building Applications*.) The easiest way to determine whether a procedure is resolved in *CtosLib.dll* is to look in a cross reference listing of *CtosLibimp.lib* you created using the **Librarian** command. Alternately, you can link the DLL with *CtosLibimp.lib* but not with any of the operating system libraries. Then, look at the map file to identify the unresolved external object module procedures.

Setting Up the Initialization Procedure

With DLLs, initialization is unique. You set up the initialization procedure in the module definition file, and the operating system runs it automatically for your DLL.

CTOS supports two types of initialization: global and instance. Global initialization runs the initialization procedure once only when the first DLL client references a DLL procedure. Instance initialization runs the procedure each time a new client first references the DLL.

Our example uses an initialization procedure called QueueInitialization. The procedure increments a global count variable called *nUsers* to keep track of the number of queue users. To maintain the count, QueueInitialization must be run with each new client when it first references the DLL. This is instance initialization.

You must specify the initialization type you want to use in the module definition file using the initialization parameter of the LIBRARY statement. To define instance initialization for the DLL, we specify the value INITINSTANCE, as shown in the exerpt from the *Queue.def* file below:

```
LIBRARY QUEUE INITINSTANCE QueueInitialization

.

.

.

EXPORTS
    QueueInitialization
    QueueAdd
    QueueDelete
    QueueStatus
```

Note also that *Queue.def* lists the QueueInitialization procedure as an export. Export procedures typically are called by clients. Although clients usually do not call initialization procedures, you must include the procedure name in this list so that the loader can find it. Otherwise the system returns the following error message:

```
QueueInitialization has been defined as an initialization
procedure, but not exported.
```

In addition to the function(s) you choose to have your DLL initialization procedure perform, the initialization procedure provides the loader with the name(s) of the procedures to be executed when the current user terminates either normally or because of an error condition.

After incrementing *nUsers*, QueueInitialization calls the CTOS request ExitListSet to define the termination procedure to be run. (See "Using a Termination Procedure.")

It is important that QueueInitialization open and lock a semaphore before it performs any of its functions. If it fails at any point, the semaphore is cleared and closed. We discuss sempahores in detail below. (See "Using Semaphores to Handle Global Data.")

After QueueInitialization establishes the termination procedure for the current client, it checks to see if the client is the first queue user. If so, it reads the queue into memory.

Note: *QueueInitialization calls the C runtime procedure C_INIT to initialize resources such as the runtime heap.*

Using a Termination Procedure

You use a termination procedure to clean up any resources allocated by a DLL on behalf of a client. You can elect of have none or as many termination procedures as your DLL requires.

Like initialization, the handling of termination is accomplished in a special way for DLL clients. You set up the termination procedure(s) to be executed through initialization, which you define in the module definition file.

Queue.c runs one termination procedure with each terminating client. QueueInitialization establishes the procedure with the call to ExitListSet, as shown below:

```
if ((erc = ExitListSet(PRIORITY_LAST|EXITLIST_ADD,
   (Pointer)QueueTermination)) != ercOk) {
```

The first parameter (PRIORITY_LAST | EXITLIST_ADD) to ExitListSet is a flag word. You use the low-order bit of the low-order byte to indicate whether you want to add or remove a termination procedure. A value of 1 for this bit adds the procedure; 0 removes it. In the *Queue.c* declarations, the add and remove functions are defined as constants, as shown below:

```
#define EXITLIST_ADD                    0x01
#define EXITLIST_REMOVE                 0x00
```

The high-order byte of the flag word sets the priority of the procedure. To define the priority of the termination procedure, our example uses the constant PRIORITY_LAST. It is defined with the value 0x9F00, which is the lowest priority (highest numerical value) in the priority range of termination procedures for general use. (For details on priority ranges, see the description of ExitListSet in the *CTOS Procedural Interface Reference Manual.*)

The operating system checks the priorities of termination procedures to determine which to run first. Say, for example, a client of *Queue.dll* is terminating. It is also a client of *Foo.dll*. *Foo.dll* happens to have defined a termination procedure with a priority of 0x2000. In this situation, the operating system runs *Foo.dll*'s termination procedure before it runs QueueTermination. In an environment where multiple DLLs are executing at the same time, it is not unlikely that there are interdependencies requiring an order to termination.

The second parameter to ExitListSet is the memory address of the termination procedure you want to add or remove. Our example specifies the address of the procedure QueueTermination, which ExitListSet is adding to the list of procedures to be run when a client terminates.

Upon termination of a client, QueueTermination decrements the global user count variable *nUsers*. If this is the last user to use the queue, the procedure calls WriteQueueFromMemory to save the queue to disk.

Note: *QueueTermination calls the C runtime procedure C_TERM to deallocate resources such as the runtime heap.*

Using Semaphores to Handle Global Data

Typically when you write applications to be run on CTOS, you can use the interprocess kernel primitives such as Request, Send, and Wait to synchronize the use of a resource by multiple users. A dynamic link library, however, requires special handling of global data among its many clients. To ensure exclusive access to the global data by each client, you need to use a CTOS semaphore.

The queue in our example is global. It is designed to be used by several clients at the same time. By necessity, it uses a CTOS system semaphore to ensure that only one user manipulates the queue data at any one time.

A semaphore ensures that the DLL gets a valid count of users to determine when to write the queue to a disk file or to read it into memory. The semaphore protects the global variable *nUsers*, which must be updated with each new user.

A semaphore also provides the DLL with valid data about the queue, such as where to add the next queue new entry for a client and how much space is left in the queue. The queue pointers *pFrontQueue* and *pEndQueue* are additional global data that must be managed by semaphores.

In the subsections below, we explain how our example DLL uses semaphores to manage its global variables.

In QueueInitialization

QueueInitialization increments the global user count *nUsers* to add the current client and to keep track of the number of DLL users.

Prior to updating *nUsers*, QueueInitialization opens and locks a system semaphore with the respective CTOS system operations SemOpen and SemLock.

In the call to SemOpen, QueueInitialization specifies the semaphore name and characteristics. The code that calls SemOpen is shown below:

```
if ((erc = SemOpen("semQueue", 8,
SEMOPEN_NORMAL, &semQueue)) != ercOk)
      ErrorExit(erc);
```

The semaphore characteristics parameter SEMOPEN_NORMAL is defined in the *Queue.c* declarations as a constant with the value 0 for mutual exclusion.

If QueueInitialization determines that this is the first user of the queue (*nUsers* equals 1), it calls ReadQueueIntoMemory to read the queue into memory. Note that the code to ReadQueueIntoMemory requires that the queue semaphore be locked before it is executed.

If, at any time, a procedure returns an error, the exit procedure clears and closes the semaphore. An example of this is shown in the code below:

```
/* Call to ExitListSet failed, clean
 * up and exit. */
      --nUsers;
      SemClear((DWord)semQueue);
      SemClose((DWord)semQueue);
      ErrorExit(erc);
```

Upon successful initialization, QueueInitialization clears the semaphore with the CTOS SemClear operation.

In QueueTermination

When a client terminates, QueueTermination decrements the global user count *nUsers*. Before doing so, however, it calls the CTOS system operation SemLock to obtain (lock) the queue semaphore.

If the semaphore is already locked by the client process (SemLock returns the error ERC_SEM_OWNER_TERMINATED), the client may be in the process of adding or deleting a queue entry. Although the code to QueueTermination does not show such a procedure for handling this error, a resilient program should provide a way for a client to complete a queue update before QueueTermination is continued.

QueueTermination does not, however, need to be concerned over whether the client process already owns the semaphore. The queue semaphore is a system semaphore. As such, it has internal management structures to ensure that it does not get relocked in this situation. (For details system semaphores, see "Semaphore Management" in the *CTOS Operating System Concepts Manual.*)

If QueueTermination determines this is the last client to use the queue (*nUsers* equals 0), it writes the queue to disk by calling WriteQueueFromMemory. Note that the call to WriteQueueFromMemory requires that the queue semaphore be locked before it is executed.

Upon termination of the current client, QueueTermination clears and closes the semaphore with the respective CTOS operations SemClear and SemClose. Closing the semaphore causes the operating system to decrement the semaphore user count. When user count reaches 0, the operating system deallocates the semaphore.

In The Export Procedures

The export procedures QueueAdd and QueueDelete manipulate the global queue pointers *pFrontQueue* and *pEndQueue* The code in these procedures cannot be executed without first obtaining (locking) the queue semaphore with the CTOS system operation SemLock. At the end of each procedure, the semaphore is cleared with the CTOS SemClear operation.

QueueAdd calls the QueueFreeSpace procedure to determine if there is enough free space to add the current entry. Becaues QueueFreeSpace calculates the amount of free space based on the global pointers *pFrontQueue* and *pEndQueue*, QueueAdd requires the semaphore for the exclusive use of these pointers.

QueueAdd continues to hold the semaphore until it adds the new entry and updates *pFrontQueue* and *pEndQueue* to new values with the IncrementQueuePtr procedure.

QueueDelete calls the FQueueEmpty procedure to determine if the queue is empty before it attempts to delete an entry. FQueueEmpty compares the value of *pFrontQueue* to *pEndQueue*. To obtain accurate results, QueueDelete obtains the semaphore before calling FQueueEmpty.

Like QueueAdd, once QueueDelete has successfully deleted an entry, it must update the queue pointers *pFrontQueue* and *pEndQueue* to new values with IncrementQueuePtr. Only when this is accomplished does QueueDelete clear the semaphore.

Setting Up Global Data Variables

The QueueInitialization procedure uses a semaphore to manage use of the following global variables:

- *pQueueMem*
- *pFrontQueue*
- *pEndQueue*
- *nUsers*

These variables are declared as global in the *Queue.c* declarations, as shown in the excerpt below:

```
#define GlobalSeg(name) \
            _based (_segname(name)) _plm
#define GlobalDataSeg  GlobalSeg("GLOBAL_SEG")
#define GlobalQueueSeg  GlobalSeg("QUEUE_SEG")

/* Define queue pointers as global variables.
 * This keeps track of the next free byte
 * in the queue. */

QueuePtr GlobalDataSeg pQueueMem;
QueuePtr GlobalDataSeg pFrontQueue;
QueuePtr GlobalDataSeg pEndQueue;

/* Define the number of users as a global
 * variable.  This determines when to
 * read/write the queue from/to disk. */

Word         GlobalDataSeg nUsers;

/* Define a global semaphore variable. */

DWord        GlobalDataSeg semQueue = 0L;

#define QUEUESIZ                        0x8000
Byte         GlobalQueueSeg Queue[QUEUESIZ];
```

The module definition file *Queue.def* defines the default data type as nonglobal (instance) data. To define instance data, you specify the value Nonshared for the Data statement. To specify global data as well and thereby override the default, you define the global segments with the Segments statement in this file. The respective *Queue.def* entries are shown below:

```
DATA MULTIPLE Nonshared
SEGMENTS
    Global_Seg Class 'Far_data' Shared
    Queue_Seg Class 'Far_data' Shared
```

Note: *Our example defines two segments for illustrative purposes only. As long as the combined length of all the variables based in the segment is less than 64K bytes (which happens to be the case), we really only needed to define one segment.*

For each global segment you specify for your DLL, you must define the segment name, for example, Global_Seg, in your source file declarations. (Refer back to the excerpt from the *Queue.c* declarations, above). The class name 'Far_data' is CTOS Microsoft C's class name for data not grouped with the automatic data segment, DGroup. (For details, see the description of the Segments statement in "Module Definition Utility," in the *CTOS Programming Utilities Reference Manual: Building Applications*.)

Setting Up Instance Data Variables

The *Queue.c* declarations define two instance data variables, *nAdds* and *nDeletes*, as shown below:

```
/*****************************************
 *** Instance Data ***
 *****************************************/
Word nAdds = 0;
Word nDeletes = 0;
```

Note: *These declarations allow the variables* nAdds *and* nDeletes *to be accessed by all modules in the DLL. In C language, these are called "global" variables. The C definition of global should not be confused with the DLL definition.*

As previously mentioned, Nonshared (instance) data is the default data type.

The operating system loads a separate copy of instance data for each DLL client. An instance data variable keeps track of data on a per user basis (in a data segment defined in the user's own memory). The respective variables *nAdds* and *nDeletes*, for example, count the number of times each client adds and deletes a queue entry. The DLL exports the procedure QueueStatus to report to the caller the number of queue entries it alone was responsible for adding and deleting.

Preparing the DLL

In this subsection, we describe a manual method of preparing (that is, compiling and building) a DLL using separate utilities executable from the Executive command line. We compare this method to an automated one employing a makefile containing all the Executive commands to accomplish the same preparation task.

The CTOS Microsoft C compiler package provides you with different options for automating the preparation process. The Programmer's Workbench (PWB), for example, is a single, integrated tool that allows you to write, debug, compile, and build C applications. You also can use the CTOS Program-Maintenance (**NMake**) utility separately from PWB to read and execute a customizable makefile of Executive commands to compile, link, and build C programs. If you use either of these utilities, you must have installed the Task Management Service and the Environment Service. (For details on all the tools and compiler options for building DLLs written in CTOS Microsoft C, see the *CTOS Microsoft C Programming and Installation Reference Manual*.)

Makefiles allow you to prepare applications in a controlled and consistent fashion. As such, they are the preferred method of doing this in real-life situations. However, we recommend that you first understand the mechanics of accomplishing the individual steps using separate Executive commands.

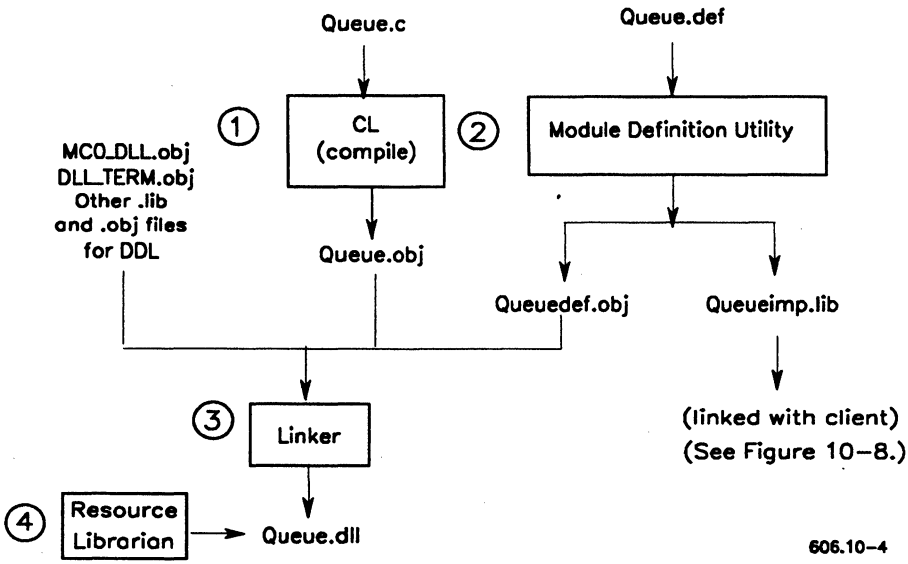
Overview of Preparing a DLL

Figure 10-4 is an overview of preparing the DLL example. It shows the following steps:

1. Compile the source file, *Queue.c*, to create *Queue.obj*.
2. Create the Dll object module, *Queuedef.obj*, and the import library, *Queueimp.lib*, with the **Module Definition** utility.
3. Link *MC0_DLL.obj*, *DLL_TERM.obj*, *Queue.obj*, and *Queuedef.obj* with any other object and library files there are for the DLL to create *Queue.dll*. (Example 10-8 shows the libraries with which this example is linked.) You can use the **Librarian** utility to organize objects and libraries.
4. Optionally add resources. You can, for example, have your DLL symbol file automatically loaded for debugging if you add it to the Debugger symbol file using the **Resource Librarian**.

(For details on how to use each of the command line utilities shown, see the *CTOS Programming Utilities Reference Manual: Building Applications*.)

Figure 10-4. Preparing the DLL



Makefile and Executive Commands

See Example 10-3. It shows the DLL makefile *Dll.mak*. The makefile automates executing the individual Executive commands for compiling and building the DLL.

To familiarize you with the general contents and to help you recognize the individual commands this file executes, we introduce a few rules for creating makefiles here. However, we leave it to you to investigate creating makefiles on your own. (For details on makefile syntax, see the section entitled "Managing Development Projects with NMAKE," in the *CTOS Microsoft C Programming and Installation Reference Manual.*)

In all makefiles, comments are preceded by the pound sign (#). For your convenience in perusing *Dll.mak*, we set up some of comments as labels to identify major subsections.

The directory specifications are the directories into which the DLL electronic source files (packaged with this documentation) are copied.

Note: *Your directories may vary somewhat from those shown. Before you run this program, be sure to make the appropriate adjustments, as necessary, to your path information.*

The makefile uses abbreviations as a shorthand representation of longer strings of data, such as path information and so forth. For example, the abbreviation MSC_INC represents the string

```
[Sys]<MscIncludes>
```

The makefile uses macros. You can identify a macro by parentheses preceded by the dollar sign ($), for example,

```
$(MSC_INC)
```

This macro replaces the abbreviation MSC_INC with the expanded string '[Sys]<MscIncludes>'.

There is an order to makefile definitions. Macros, for example, must be defined before they can be used. The commands that use them can be placed in the file in any order following the macro definitions.

In the paragraphs below, we look at the steps for compiling and building the DLL. We also show you the commands and their parameter values as you would enter them into the Executive command forms.

Compiling Queue.c

The *Dll.mak* makefile syntax for compiling *Queue.c* to produce *Queue.obj* is shown below:

```
queue.obj:queue.c
    Delete\n $*.obj\g
    CL\n @<<[Scr]<$$>cFlags.fls
-c -ML -Gs -I$(MSC_INC)
<<
    \n $*.c >$*.lst\g
```

These lines show two types of information: the source file to operate on (*Queue.c*) and resulting the file produced (*Queue.obj*). The respective makefile names for these parts are dependent and target.

Following the target/dependents line, you see the commands the makefile executes to compile *Queue.c* and produce *Queue.obj*. The syntax $* is a predefined macro that evaluates to the target's base name (*Queue*). For example, $.lst evaluates to *Queue.lst*.

The steps below accomplish this same compilation procedure using Executive commands:

1. Delete *Queue.obj*:

```
Delete
 File list                Queue.obj
 [Confirm each?]
```

2. Copy the compiler options to the temporary file *[Scr]<$>Flag.tmp*:

```
Copy
 File from                [Kbd]
 File to                  [Scr]<$>Flag.tmp
 [Overwrite OK?]          Yes
 [Confirm each?]
```

3. Type in the following characters, separating each option with a space; then press **FINISH**. (Do not insert any carriage returns.)

```
-c -ML -Gs -I[Sys]<MscIncludes>
```

4. Compile *Queue.c* with the CTOS Microsoft C compiler command **CL**:

```
CL
 [Args]                   @[Scr]<$$>Flag.tmp
 [Args]                   Queue.c>Queue.lst
```

Creating the Object Module and Import Library

The makefile syntax for creating the object file *Queuedef.obj* and the import library *Queueimp.lib*, respectively, is shown below:

```
queuedef.obj: queue.def
     Module Definition\n\
          Queue.Def\n\
          QueueDef.obj\g

queueimp.Lib: queue.def
     Module Definition\n\
          Queue.Def\n\
          \n\
          QueueImp.Lib\g
```

From the Executive command line, you can execute the **Module Definition** command to create the object module and import library. You fill out the command form as shown below:

```
Module Definition
 Imput file                    Queue.def
 [Object module]               Queuedef.obj
 [Import library]              Queueimp.lib
 [List file]
 [Suppress warnings?]
 [Suppress ordinals?]
 [Uppercase?]
```

Linking The DLL Object and Library Files

The makefile syntax for linking the DLL object and library files is shown below:

```
queue.dll: queue.obj queuedef.obj
     Link V8\n @<<[scr]<$$>link.fls
$(MSC_LIBS)MC0_DLL.obj
$(MSC_LIBS)DLLTERM.obj
Queue.obj
QueueDef.obj
<<
     \n Queue.dll\n\n\n\n\
          \n 0 131072\n 0 0\n protected\n \
        $(VERSION)\n (Dll)\n\n\n\n\n \
        LinkerConfig.sys\g \
     Resource Librarian\nQueue.Dll\n\
          Debugger.SymbolFile/\
          Queue.Sym\n\n\n\
          ResourceLibConfig.sys\g
```

The steps below accomplish this same linking procedure using Executive commands:

1. Copy the DLL object files to the temporary file *@[Scr]<$>Link.fls*:

```
Copy
 File from              [Kbd]
 File to                @[Scr]<$>Link.fls
 [Overwrite OK?]        Yes
 [Overwrite OK?]
```

2. Type in the following object file names. Separate each name with a space; then press **FINISH**. (Do not insert any carriage returns.)

```
[Sys]<MscLib>MC0_DLL.obj
[Sys]<MscLib>MC0_DLLTERM.obj
Queuedef.obj
Queue.obj
```

3. Link the object and library files, as shown below, to create *Queue.dll*:

```
Link V8
 Object modules                 @[Scr]Link.fls
 Run file                       Queue.dll
 [Map file]
 [Publics?]
 [Line numbers?]
 [Stack, Dgroup head size]
 [Max array, data]              0 131072
 [Min array, data]              0 0
 [Run file mode]                protected
 [Version]
 [Libraries]                    (Dll)
 [DS allocation?]
 [Symbol file]
 [Copyright notice?]
 [File to append]
 [Linker config file]           LinkerConfig.sys
```

The library files with which we link are in *LinkerConfig.sys*. (See Example 10-8.)

The makefile syntax for adding the DLL symbols to the Debugger symbol file is shown below:

```
Resource Librarian\nQueue.Dll\n\
          Debugger.SymbolFile/\
          Queue.Sym\n\n\n\
          ResourceLibConfig.sys\g
```

From the Executive command line, you can (optionally) execute the **Resource Librarian** command to add the queue symbols. You fill out the command form as shown below:

```
Resource Librarian
 Run or res file        Queue.dll
 [Resources to add]     Debugger.SymbolFile/Queue.sym
 [Resources to delete]
 [Resources to extract]
 [Resource config file] ResourceLibConfig.sys
```

The file *ResourceLibConfig.sys* is packaged with the Standard Software Development Utilities. Its contents are not modified.

Using the DLL

The WriteQueue and ReadQueue programs are simple examples of how to access the DLL export procedures. WriteQueue adds new queue entries while ReadQueue deletes them.

Client Source Files

The client program consists of the source files listed below:

File Name	Contents	Shown in
Queue.h	Program-specific headers	Example 10-4
ReadQueue.c	Client source file	Example 10-5
WriteQueue.c	Client source file	Example 10-6
Clients.mak	Clients makefile	Example 10-7

These listings are presented at the end of this section. To run the client programs, we recommend that you use the electronic versions of the files on the diskettes packaged with the Standard Software Development Utilities rather than typing the contents of the listings shown in this document.

You need to link the client programs with the CTOS Microsoft C libraries listed below. (See the CTOS Microsoft C documentation listed in "Supporting Documentation," for details.)

- *LLIBCP.lib*
- *LLIBFA.lib*
- *LIBCEXT.lib*

Our example lists these libraries in the Linker configuration file *LinkerConfig.sys*. See Example 10-8 for the contents of this file.

If you plan to use the CTOS Microsoft C **NMake** utility to build the client programs, you must have installed the Task Management Service and the Environment Service, which are packaged with CTOS Microsoft C.

Reading Queue Entries

ReadQueue.run reads the queue until it is empty.

Writing Queue Entries

WriteQueue.run writes the following four entries to the queue:

- Now is the time
- for all good men
- to come to the aid of their country.
- again

The program presents the user with the four options below:

Option	Description
Blast and wait	Writes the four entries repeatedly. If the queue fills up, it waits until there is room and then continues to fill it.
Blast and stop	Writes the four entries repeatedly until the queue fills up. Then it terminates.
Write one per **RETURN**	Writes one entry to the queue when you press the **RETURN** key.
End	Terminates and returns to the Executive.

We provide these options so you can see that the global and instance variables really work. You can run *WriteQueue.run* in one context and *ReadQueue.run* in another. You can also clone *WriteQueue.run*, changing the strings and executing each cloned run file in a different context. Have fun with it.

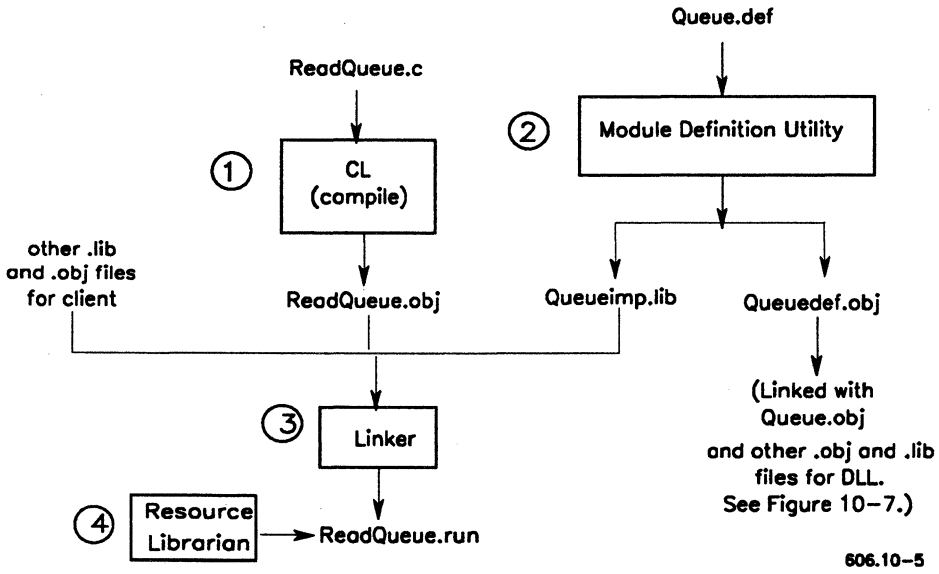
Preparing the Client Source Files

Figure 10-5 illustrates preparing a client. The steps to create the *ReadQueue.run* and *WriteQueue.run* clients are as follows:

1. Compile the source file *ReadQueue.c* to create *ReadQueue.obj*.
2. Link *ReadQueue.run*.
3. Optionally add resources.
4. Compile the source file *WriteQueue.c* to create *WriteQueue.obj*.
5. Link *WriteQueue.run*.
6. Optionally add resources.

Obtain the import library *Queueimp.lib* created for the Dll. (For details on how to prepare an import library, see "Preparing the DLL.") *Queueimp.lib* is linked with the other object files for the client to create *ReadQueue.run* and *WriteQueue.run*. (You can use the **Librarian** utility to organize objects and libraries.)

Figure 10-5. Preparing the ReadQueue Client



Client Makefile and Executive Commands

You can set up a client makefile to contain all the commands required to prepare your client program. In the description below, we show you the makefile syntax and the corresponding Executive commands to build *ReadQueue.run*. The makefile in Example 10-7 actually builds both clients. In each case, the procedure is the same.

Compiling ReadQueue.c

The makefile syntax for compiling *ReadQueue.c* is shown below:

```
ReadQueue.obj:ReadQueue.c
    Delete\n $*.obj\g
    CL\n @<<[Scr]<$$>cFlags.fls
-c -ALu -Od -G2s -Zp -W3 -I$(MSC_INC)
<<
    \n $*.c >$*.lst\g
```

The steps below accomplish this same compilation procedure using Executive commands:

1. Delete *ReadQueue.obj*:

```
Delete
 File list                    ReadQueue.obj
 [Confirm each?]
```

2. Copy the compiler options to the temporary file *[Scr]<$>Flag.tmp*:

```
Copy
 File from                    [Kbd]
 File to                      [Scr]<$>Flag.tmp
 [Overwrite OK?]              Yes
 [Overwrite OK?]
```

3. Type in the following characters, separating each option with a space; then press **FINISH**. (Do not insert any carriage returns.)

```
-c -ALu -Od -G2s -Zp -W3 -I[Sys]<MscIncludes>
```

4. Compile *ReadQueue.c* with the CTOS Microsoft C compiler command **CL**:

```
CL
 [Args]                       @[Scr]<$>Flag.tmp
 [Args]                       ReadQueue.c >ReadQueue.lst
```

Linking the Client Object and Library Files

The makefile syntax for linking the object and library files to create *ReadQueue.run* is shown below:

```
ReadQueue.Run:ReadQueue.Obj
     Link V8\n @<<[Scr]<$$>link.fls
$(MSC_LIBS)mc0.obj
ReadQueue.obj
<<
     \n ReadQueue.Run\n\n\n\
    \n8192\n 0 131072\n 0 0\n \
    protected\n $(VERSION)\n\
    (Client)\n\n\n\n\n \
    LinkerConfig.sys\g
```

The steps below accomplish this same linking procedure using Executive commands:

1. Copy the client object files to a file called *@[Scr]<$>Link.fls*:

```
Copy
 File from                  [Kbd]
 File to                    @[Scr]<$>Link.fls
 [Overwrite OK?]            Yes
 [Confirm each?]
```

2. Type in the following object file names. Separate each name with a space; then press **FINISH**. (Do not insert any carriage returns.)

```
[Sys]<MscLib>MC0.obj
 ReadQueue.obj
```

Note: *The object file* MC0.obj *is a C startup file required of all non-DLL run files written with main(). It is packaged with CTOS Microsoft C.*

3. Link the object and library files, as shown below, to create *ReadQueue.run*:

```
Link V8
 Object modules                          @[Scr]<$>Link.fls
 Run file                                    ReadQueue.run
 [Map file]
 [Publics?]
 [Line numbers?]
 [Stack, Dgroup head size]                            8192
 [Max array, data]                             0 131072
 [Min array, data]                                  0 0
 [Run file mode]                               protected
 [Version]                          1.0-%D|!*o!/!*d!/!yy!-
                                              !*t!:!0m!|
 [Libraries]                                    (Client)
 [DS allocation?]
 [Symbol file]
 [Copyright notice?]
 [File to append]
 [Linker config file]                      LinkerConfig.sys
```

See Example 10-8 for the list of library files in *LinkerConfig.sys* that are linked with the clients.

Adding Resources

The makefile syntax for adding the client symbols to the Debugger symbol file is shown below:

```
Resource Librarian\nReadQueue.Run\n\
         Debugger.SymbolFile/\
         ReadQueue.Sym\n\n\n\
         ResourceLibConfig.sys\g
```

From the Executive command line, you can execute the **Resource Librarian** command to add the queue symbols. You fill out the command forma as shown below:

```
Resource Librarian
 Run or res file                    ReadQueue.run
 [Resources to add]                 Debugger.SymbolFile/
                                    ReadQueue.sym
 [Resources to delete]
 [Resources to extract]
 [Resource config file]             ResourceLibConfig.sys
```

DLL Program Listings

Example 10-1. Queue.c

```
/*******************************************
 * Filename:     Queue.c
 * Author:
 * Date:         3/1/93
 * Compiler:     CTOS Microsoft C 6.1
 *
 * This file contains the C source code for
 * the DLL.  It must be compiled in large
 * model.  See the makefile shown in
 * Example 10-3 for the compiler options
 * used to compile Queue.c.
*******************************************/

#include <stdio.h>
#include <errno.h>
#include <stdlib.h>
#include <memory.h>
#include <sys\types.h>
#include <sys\stat.h>

typedef unsigned short int
                          Word;
typedef unsigned long     DWord;
typedef unsigned char     Byte;
typedef unsigned char     Flag;
typedef void _far *       Pointer;
typedef Word              ErcType;
typedef Word              FhType;

typedef DWord             lfaType;
typedef Byte _far *       QueuePtr;
typedef QueuePtr _far *   QueuePtrPtr;

#define APIENTRY          _plm _far _loadds
#define CTOSENTRY         _plm _far
#define FALSE             (0==1)
```

Example 10-1. Queue.c (cont.)

```
#define ercOk                   0
#define FIRST_ERC               15500
#define ERC_QUEUE_FULL          FIRST_ERC + 0
#define ERC_QUEUE_EMPTY         FIRST_ERC + 1
#define ERC_BUFFER_TOO_SMALL    FIRST_ERC + 2

#define ERC_SEM_OWNER_TERMINATED      13956

#define EXITLIST_ADD                   0x01
#define EXITLIST_REMOVE                0x00
#define PRIORITY_LAST                 0x9F00
#define WAITFOREVER                      -1

ErcType CTOSENTRY ExitListSet(Word,Pointer);
void     CTOSENTRY ErrorExit(Word);

#define SEMOPEN_NORMAL                    0

ErcType CTOSENTRY SemLock(DWord, Word);
ErcType CTOSENTRY SemClear(DWord);
ErcType CTOSENTRY SemClose(DWord);
ErcType CTOSENTRY SemOpen (Pointer, Word,
                           Word, Pointer);

typedef unsigned short far  *PUSHORT;
#define OFFSETOF(p)      (((PUSHORT)&(p))[0])

void _far _pascal C_INIT(void);
void _far _pascal C_TERM(void);


/*****************************************
 *** Global Data ***
 *****************************************/
/* To place segment into global data also
 * requires the following statement in the
 * Queue.def file:
 * SEGMENTS GLOBAL_SEG CLASS 'FAR_DATA' SHARED
 * Note that the class defaults to
 * 'FAR_DATA' */
```

Example 10-1. Queue.c (cont.)

```
#define GlobalSeg(name) \
              _based (_segname(name)) _plm

#define GlobalDataSeg  GlobalSeg("GLOBAL_SEG")
#define GlobalQueueSeg  GlobalSeg("QUEUE_SEG")

/* Define queue pointers as global variables.
 * This keeps track of the next free byte
 * in the queue. */

QueuePtr GlobalDataSeg pQueueMem;
QueuePtr GlobalDataSeg pFrontQueue;
QueuePtr GlobalDataSeg pEndQueue;

/* Define the number of users as a global
 * variable.  This determines when to
 * read/write the queue from/to disk. */

Word          GlobalDataSeg nUsers;

/* Define a global semaphore variable. */

DWord         GlobalDataSeg semQueue = 0L;

#define QUEUESIZ                          0x8000
Byte          GlobalQueueSeg Queue[QUEUESIZ];

/*****************************************
 *** Instance Data ***
 *****************************************/
Word nAdds = 0;
Word nDeletes = 0;
```

Example 10-1. Queue.c (cont.)

```
/*****************************************
 *** Procedures ***
 *****************************************/

void IncrementQueuePtr(
                    QueuePtrPtr ppQueueEntry)
{
   if (OFFSETOF(*ppQueueEntry) >= QUEUESIZ-1)
       OFFSETOF(*ppQueueEntry) =
                          OFFSETOF(pQueueMem);
   else OFFSETOF(*ppQueueEntry)++;

} /* IncrementQueuePtr */

Word QueueUsedSpace(void)
{
   if (pFrontQueue == pEndQueue)
      return 0;
   if (pEndQueue > pFrontQueue)
      return pEndQueue - pFrontQueue;
   else
      return QUEUESIZ - OFFSETOF(pFrontQueue)
                + OFFSETOF(pEndQueue);
}

Word QueueFreeSpace(void)
{
/* Always leave at least one byte free.
 * Otherwise "empty" and "full"look
 * remarkably similar. */

  return QUEUESIZ - QueueUsedSpace() - 1;

}

Flag FQueueEmpty(void)
{
  return (Flag)(pFrontQueue == pEndQueue);

}
```

Example 10-1. Queue.c (cont.)

```
ErcType ReadQueueIntoMemory(void)

{
Word      erc;
long      EOFPointer;

FILE *fp;
int   fh;
int   i;

pFrontQueue = pQueueMem;
pEndQueue   = pQueueMem;

fp = fopen("[Scr]<$>Queue.tmp", "rb");
if (fp == NULL) {
      erc = _doserrno;
    if (erc != ENOENT)
        return erc;
    return ercOk;
   }

fh = fileno(fp);
EOFPointer = filelength(fh);
if (EOFPointer == -1) {
    erc = _doserrno;
    fclose(fp);
    return erc;
   }

i = fread(pFrontQueue, 1, EOFPointer, fp);
if (i != EOFPointer || ferror(fp)) {
    fclose(fp);
      return -1;
   }

pEndQueue = pEndQueue + EOFPointer;

fclose(fp);
remove("[Scr]<$>Queue.tmp");

return ercOk;
}
```

Example 10-1. Queue.c (cont.)

```
/* Queue semaphore should be locked before
 * calling this procedure. */

ErcType WriteQueueFromMemory(void)
{
Word      erc;
Word      i;
int       fh;
FILE      *fp;
Word      cbTotal;


if (FQueueEmpty())
    return ercOk;

cbTotal = QueueUsedSpace();

fp = fopen("[Scr]<$>Queue.tmp", "w");
if (fp == NULL) {
    erc = _doserrno;
    return erc;
   }

i = fwrite(pFrontQueue, 1, cbTotal, fp);
if (i != cbTotal) {
    erc = _doserrno;
    return erc;
   }

fclose(fp);

return ercOk;

}
```

Example 10-1. Queue.c (cont.)

```
void APIENTRY QueueTermination(ErcType
TerminationCode)

{
ErcType erc;

/* Note:  If the semaphore was locked by this
 * process when termination was called, then
 * there is a chance that this process was in
 * the middle of updating the queue.
 * Terminating at this point could
 * corrupt the queue.  A resilient program
 * would need to complete the update before
 * terminating. */


erc = SemLock(semQueue, WAITFOREVER);
if (erc != ercOk)
      return;

/* If this is the last active user of queue.dll,
 * save the queue to disk. */
if (--nUsers == 0)
      erc = WriteQueueFromMemory();

SemClear((DWord)semQueue);
SemClose(semQueue);

C_TERM();

}
```

Example 10-1. Queue.c (cont.)

```
/* QueueInitialization increments the
 * user count and if it is the "first" user,
 * then it reads the queue from disk. */

void APIENTRY QueueInitialization(void)
{
ErcType erc;


   C_INIT();

   if ((erc = SemOpen("semQueue", 8,
   SEMOPEN_NORMAL, &semQueue)) != ercOk)
      ErrorExit(erc);

   if ((erc = SemLock(semQueue, WAITFOREVER))
   != ercOk)
      if (erc != ERC_SEM_OWNER_TERMINATED)
         ErrorExit(erc);

   nUsers++;
   if ((erc = ExitListSet
   (PRIORITY_LAST|EXITLIST_ADD,
   (Pointer)QueueTermination)) != ercOk) {

      /* Call to ExitListSet failed, clean
       * up and exit. */
      --nUsers;
      SemClear((DWord)semQueue);
      SemClose((DWord)semQueue);
      ErrorExit(erc);
      }
```

Example 10-1. Queue.c (cont.)

```
/* If this is the first user of the queue
 * dynamic link library,then read the queue
 * from disk.*/

   if (nUsers == 1) {
      pQueueMem = Queue;
      if (erc = ReadQueueIntoMemory()) {
         SemClose((DWord)semQueue);
         ErrorExit(erc);
         }
      }

   SemClear((DWord)semQueue);

}


ErcType APIENTRY QueueAdd(pbQueueEntry,
cbQueueEntry)

QueuePtr       pbQueueEntry;
Byte           cbQueueEntry;

{
Word    i;
ErcType erc;

   if ((erc = SemLock(semQueue, WAITFOREVER))
   != ercOk)
      if (erc != ERC_SEM_OWNER_TERMINATED)
         ErrorExit(erc);

/*  The first byte of a queue entry
 * contains the size of the entry. */

   if (QueueFreeSpace() <
                  (Word)(cbQueueEntry + 1)) {
      SemClear((DWord)semQueue);
      return ERC_QUEUE_FULL;
      }
```

Example 10-1. Queue.c (cont.)

```
   *pEndQueue = cbQueueEntry;
   IncrementQueuePtr(&pEndQueue);

   i = 0;
   while (i < cbQueueEntry) {
      *pEndQueue = pbQueueEntry[i++];
      IncrementQueuePtr(&pEndQueue);
      }

   if ((erc = SemClear((DWord)semQueue))
   != ercOk)
      return erc;

   nAdds++;
   return ercOk;

}


ErcType APIENTRY QueueDelete(pbQueueEntry,
     cbQueueEntryMax, pcbQueueEntryRet)

QueuePtr       pbQueueEntry;
Byte           cbQueueEntryMax;
Byte          *pcbQueueEntryRet;

{
Byte    cbTotal;
Word    i;
ErcType erc;


   if ((erc = SemLock(semQueue,
   WAITFOREVER)) != ercOk)
      if (erc != ERC_SEM_OWNER_TERMINATED)
         ErrorExit(erc);
```

Example 10-1. Queue.c (cont.)

```
    if (FQueueEmpty()) {
        *pcbQueueEntryRet = 0;
        SemClear((DWord)semQueue);
        return ERC_QUEUE_EMPTY;
        }

/* pFrontQueue points to the next queue
 * entry.  The first byte contains the size
 * of the entry. */

    if (*pFrontQueue > cbQueueEntryMax) {
        SemClear((DWord)semQueue);
        return ERC_BUFFER_TOO_SMALL;
        }

    cbTotal = *pFrontQueue;
    *pcbQueueEntryRet = cbTotal;
    IncrementQueuePtr(&pFrontQueue);

    i = 0;
    while (i < cbTotal) {
        pbQueueEntry[i++] = *pFrontQueue;
        IncrementQueuePtr(&pFrontQueue);
        }

    if ((erc = SemClear((DWord)semQueue))
    != ercOk)
        return erc;

  nDeletes++;
  return ercOk;

}
```

Example 10-1. Queue.c (cont.)

```
ErcType APIENTRY QueueStatus(pnDeletes,
pnAdds,pcbFree,pcbUsed)
Word *pnDeletes;
Word *pnAdds;
Word *pcbFree;
Word *pcbUsed;
{

*pnDeletes = nDeletes;
*pnAdds = nAdds;
*pcbFree = QueueFreeSpace();
*pcbUsed = QueueUsedSpace();

return ercOk;
}
```

Example 10-2. Queue.def

```
; ---------------------------------------- ;
; Filename:     Queue.def
; Author:
; Date:         5/28/93
;
; This file is the module definition
; file for the queue DLL.
; ---------------------------------------- ;

LIBRARY QUEUE INITINSTANCE QueueInitialization

DESCRIPTION 'CTOS QUEUE EXAMPLE'

DATA MULTIPLE Nonshared

SEGMENTS
    Global_Seg Class 'Far_data' Shared
    Queue_Seg Class 'Far_data' Shared

EXPORTS
    QueueInitialization
    QueueAdd
    QueueDelete
    QueueStatus
```

Example 10-3. Dll.mak

```
# ---------------------------------------- #
# Filename:      Dll.mak
# Author:
# Date:          3/1/93
# Compiler:      CTOS Microsoft C 6.1
#
# The Environment Service and Task Management
# Service must be installed before NMAKE can
# process this file.
#
# This file is the makefile for compiling
# and building the queue DLL.
#
# Invoke:
#  NMAKE
#    [Args] -f dll.mak
#    [    ] queue.dll queueimp.lib
# ---------------------------------------- #

# ---------------------------------------- #
# Default version string
# ---------------------------------------- #
VERSION = 1.0-%D|!*o!/!*d!/!yy!-!*t!:!0m!|

# ---------------------------------------- #
# Directory Specifications
# ---------------------------------------- #

MSC_INC  = [Sys]<MscIncludes>
CTOS_DIR = [Sys]<Sys>
MSC_LIBS = [Sys]<MscLib>

# ---------------------------------------- #
# Create Queue.dll
# ---------------------------------------- #

queue.dll: queue.obj queuedef.obj
    Link V8\n @<<[scr]<$$>link.fls
$(MSC_LIBS)MC0_DLL.obj
$(MSC_LIBS)DLLTERM.obj
Queue.obj
QueueDef.obj
<<
    \n Queue.dll\n\n\n\n\
        \n 0 131072\n 0 0\n protected\n \
       $(VERSION)\n (Dll)\n\n\n\n\n \
       LinkerConfig.sys\g \
```

Example 10-3. Dll.mak (cont.)

```
    Resource Librarian\nQueue.Dll\n\
          Debugger.SymbolFile/\
          Queue.Sym\n\n\n\
          ResourceLibConfig.sys\g

# ------------------------------------- #
# Compile Queue.c
# ------------------------------------- #
queue.obj:queue.c
    Delete\n $*.obj\g
    CL\n @<<[Scr]<$$>cFlags.fls
-c -ML -Gs -I$(MSC_INC)
<<
    \n $*.c >$*.lst\g

# ------------------------------------- #
# Create Def Object Module
# ------------------------------------- #
queuedef.obj: queue.def
    Module Definition\n\
          Queue.Def\n\
          QueueDef.obj\g

# ------------------------------------- #
# Create Import Library
# ------------------------------------- #
queueimp.Lib: queue.def
    Module Definition\n\
          Queue.Def\n\
          \n\
          QueueImp.Lib\g
```

Example 10-4. Queue.h

```
/*****************************************
 * Filename:      Queue.h
 * Author:
 * Date:          5/28/93
 * Compiler:      CTOS Microsoft C 6.1
 *
 * This file is the header file for ReadQueue.c
 * and WriteQueue.c.
 *****************************************/

typedef unsigned char          Byte;
typedef unsigned short int     Word;
typedef Word                   ErcType;

typedef Byte _far *             QueuePointer;
#define APIENTRY               _plm _far

#define ercOk                                  0

extern ErcType APIENTRY QueueAdd(QueuePointer
pbQueueEntry, Byte cbQueueEntry);

extern ErcType APIENTRY QueueDelete
(QueuePointer pbQueueEntry,Byte
cbQueueEntryMax, Byte *pcbQueueEntryRet);

extern ErcType APIENTRY QueueStatus(Word
*pnDeletes,  Word *pnAdds, Word *pcbFree,
Word *pcbUsed);
```

Example 10-5. ReadQueue.c

```
/*******************************************
 * Filename:      ReadQueue.c
 * Author:
 * Date:          3/1/93
 * Compiler:      CTOS Microsoft C 6.1
 *
 * This file contains the C source code for
 * a client of the DLL.  The client calls
 * the DLL export procedures QueueStatus
 * and QueueDelete.
 *******************************************/

#include "queue.h"
#include <stdio.h>

int main() {

ErcType erc;
Byte    String[81];
Byte    DataRet;
Word    nDeletes;
Word    nAdds;
Word    cbFree;
Word    cbUsed;

if ((erc = QueueStatus(&nDeletes,&nAdds,
&cbFree,&cbUsed)) != ercOk)
    return erc;
else
   printf("nDeletes = %d; nAdds = %d;
   cbFree = %d; cbUsed = %d\n\n", nDeletes,
   nAdds, cbFree, cbUsed);
while (erc == ercOk) {
   if ((erc = QueueDelete(String,80,&DataRet))
   == ercOk) {
      String[DataRet] = '\0';
      printf("%s\n",String);
      }
   }
if ((erc = QueueStatus(&nDeletes,&nAdds,
&cbFree,&cbUsed)) != ercOk)
    return erc;
else
    printf("nDeletes = %d; nAdds = %d;
    cbFree = %d; cbUsed = %d\n\n", nDeletes,
    nAdds, cbFree, cbUsed);

return ercOk;
}
```

Example 10-6. WriteQueue.c

```
/*****************************************
 * Filename:     WriteQueue.c
 * Author:
 * Date:         3/1/93
 * Compiler:     CTOS Microsoft C 6.1
 *
 * This file contains the C source code for
 * a client of the DLL.  The client calls
 * the DLL export procedures QueueAdd and
 * QueueStatus.
 *****************************************/

#include <stdio.h>

#include "queue.h"

#define CTOSENTRY       _plm _far
ErcType CTOSENTRY Beep(void);
ErcType CTOSENTRY Delay(Word);

int main() {

ErcType erc;
Word    nDeletes;
Word    nAdds;
Word    cbFree;
Word    cbUsed;
Byte    b;
Byte    b1;

while (1) {
printf("\nChoose a mode\n\n");
printf("\t1 - Blast and wait\n");
printf("\t2 - Blast and stop\n");
printf("\t3 - Write one per RETURN\n");
printf("\t4 - End\n");
b = (Byte)getchar();
printf("\n");

switch (b) {
    case '1':
        while (1) {
w1:
            erc = QueueAdd("Now is the time",
            15);
            if (erc != ercOk) {
                Delay(10);
                goto w1;
               }
```

Example 10-6. WriteQueue.c (cont.)

```
w2:
            erc = QueueAdd("for all good men",
            16);
            if (erc != ercOk) {
                Delay(10);
                goto w2;
               }
w3:
            erc = QueueAdd("to come to the
            aid of their country.",36);
            if (erc != ercOk) {
                Delay(10);
                goto w3;
               }
w4:
            erc = QueueAdd("Again", 5);
            if (erc != ercOk) {
                Delay(10);
                goto w4;
               }
          }
        if ((erc = QueueStatus(&nDeletes,&nAdds,
        &cbFree,&cbUsed)) != ercOk)
            return erc;
        else
            printf("nDeletes = %d; nAdds = %d;
            cbFree = %d; cbUsed = %d\n",
            nDeletes, nAdds, cbFree, cbUsed);
        break;
    case '2':
        erc = ercOk;
        while (1) {
            erc = QueueAdd("Now is the time",
            15);
            if (erc != ercOk)
                break;
            erc = QueueAdd("for all good men",
            16);
            if (erc != ercOk)
                break;
            erc = QueueAdd("to come to the aid
            of their country.",36);
            if (erc != ercOk)
                break;
            erc = QueueAdd("Again", 5);
            if (erc != ercOk)
                break;
          }
```

Example 10-6. WriteQueue.c (cont.)

```
        if ((erc = QueueStatus(&nDeletes,&nAdds,
        &cbFree,&cbUsed)) != ercOk)
            return erc;
        else
            printf("nDeletes = %d; nAdds = %d;
            cbFree = %d; cbUsed = %d\n",
            nDeletes, nAdds, cbFree, cbUsed);
        break;
    case '3':
        while (1) {
x1:
            printf("\nPress RETURN to write
            to the queue, FINISH to quit");
            b1 = (Byte)getchar();
            if (b1 == 4)
                break;
            erc = QueueAdd("Now is the time",
            15);
            if (erc != ercOk) {
                printf("Error Number = %d\n",
                erc);
                Delay(10);
                goto x1;
               }
x2:
            printf("\nPress RETURN to write to
            the queue, FINISH to quit");
            b1 = (Byte)getchar();
            if (b1 == 4)
                break;
            erc = QueueAdd("for all good men",
            16);
            if (erc != ercOk) {
                printf("Error Number = %d\n",
                erc);
                Delay(10);
                goto x2;
               }
x3:
            printf("\nPress RETURN to write
            to the queue, FINISH to quit");
            b1 = (Byte)getchar();
            if (b1 == 4)
                break;
            erc = QueueAdd("to come to the
            aid of their country.",36);
```

Example 10-6. WriteQueue.c (cont.)

```
                if (erc != ercOk) {
                    printf("Error Number = %d\n",
                    erc);
                    Delay(10);
                    goto x3;
                   }
x4:
                printf("\nPress RETURN to write to
                the queue, FINISH to quit");
                b1 = (Byte)getchar();
                if (b1 == 4)
                    break;
                erc = QueueAdd("Again", 5);
                if (erc != ercOk) {
                    printf("Error Number = %d\n",
                    erc);
                    Delay(10);
                    goto x4;
                   }
               }
            if ((erc = QueueStatus(&nDeletes,&nAdds,
            &cbFree,&cbUsed)) != ercOk)
                return erc;
            else
                printf("nDeletes = %d; nAdds = %d;
                cbFree = %d; cbUsed = %d\n",
                nDeletes, nAdds, cbFree, cbUsed);
            break;
        case '4':
            return 0;
        default:
            Beep();
            break;
       }

    }

    }
```

Example 10-7. Clients.mak

```
# ---------------------------------------- #
# Filename:      Clients.mak
# Author:
# Date:          3/1/93
# Compiler:      CTOS Microsoft C 6.1
#
# The Environment Service and Task Manage-
# ment Service must be installed before
# NMAKE canprocess this file.
#
# This file is the makefile for the two
# clients of the DLL.  They call the DLL
# export procedures QueueAdd, QueueDelete,
# and QueueStatus.
#
# Invoke:
#  NMAKE
#    [Args] -f clients.mak
#    [    ] readqueue.run writequeue.run
# ---------------------------------------- #

# ---------------------------------------- #
# Default version string
# ---------------------------------------- #
VERSION = 1.0-%D|!*o!/!*d!/!yy!-!*t!::!0m!|

# ---------------------------------------- #
# Directory Specifications
# ---------------------------------------- #

MSC_INC  = [Sys]<MscIncludes>
CTOS_DIR = [Sys]<Sys>
MSC_LIBS = [Sys]<MscLib>
```

Example 10-7. Clients.mak (cont.)

```
# ---------------------------------------- #
# Create ReadQueue.run
# ---------------------------------------- #
ReadQueue.Run:ReadQueue.Obj
      Link V8\n @<<[Scr]<$$>link.fls
$(MSC_LIBS)mc0.obj
ReadQueue.obj
<<
      \n ReadQueue.Run\n\n\n\
    \n8192\n 0 131072\n 0 0\n \
    protected\n $(VERSION)\n\
    (Client)\n\n\n\n\n \
    LinkerConfig.sys\g \
      Resource Librarian\nReadQueue.Run\n\
         Debugger.SymbolFile/\
         ReadQueue.Sym\n\n\n\
         ResourceLibConfig.sys\g

# ---------------------------------------- #
# Create WriteQueue.run
# ---------------------------------------- #
WriteQueue.Run:WriteQueue.Obj
      Link V8\n @<<[Scr]<$$>link.fls
$(MSC_LIBS)mc0.obj
WriteQueue.obj
<<
      \n WriteQueue.Run\n\n\n\
    \n8192\n 0 131072\n 0 0\n \
    protected\n $(VERSION)\n\
    (Client)\n\n\n\n\n \
    LinkerConfig.sys\g \
      Resource Librarian\nWriteQueue.Run\n\
         Debugger.SymbolFile/\
         WriteQueue.Sym\n\n\n\
         ResourceLibConfig.sys\g
```

Example 10-7. Clients.mak (cont.)

```
# ------------------------------------- #
# Compile ReadQueue.c
# ------------------------------------- #
ReadQueue.obj:ReadQueue.c
      Delete\n $*.obj\g
      CL\n @<<[Scr]<$$>cFlags.fls
-c -ALu -Od -G2s -Zp -W3 -I$(MSC_INC)
<<
      \n $*.c >$*.lst\g

# ------------------------------------- #
# Compile WriteQueue.c
# ------------------------------------- #
WriteQueue.obj:WriteQueue.c
      Delete\n $*.obj\g
      CL\n @<<[Scr]<$$>cFlags.fls
-c -ALu -Od -G2s -Zp -W3 -I$(MSC_INC)
<<
      \n $*.c >$*.lst\g
```

Example 10-8. LinkerConfig.sys

```
:LibraryReference:Dll
:LibraryFile:[Sys]<MscLib>LLIBCDLL.LIB
:LibraryFile:[Sys]<Sys>Ctos.lib
:LibraryFile:[Sys]<Sys>Systemimp.lib

:LibraryReference:Client
:LibraryFile:QueueImp.lib
:LibraryFile:[Sys]<MscLib>LLIBCP.LIB
:LibraryFile:[Sys]<MscLib>LLIBFA.LIB
:LibraryFile:[Sys]<MscLib>LIBCEXT.LIB
:LibraryFile:[Sys]<Sys>Ctos.lib
:LibraryFile:[Sys]<Sys>CtosToolKit.lib
:LibraryFile:[Sys]<Sys>ENLS.lib
```

Index

8274 serial controller, 9-3
8530 serial controller, 9-3
  EOF reporting, 9-8

A

a1-characters,
  CD-ROM files, 17-60
Adaptive Differential Pulse Code Modulation (ADPCM), 14-13
adding DLL client resources, 10-34
adding DLL resources, 10-26
AddQueue, 12-7
AddQueueEntry, 12-7
address space protection, 2-12
aliasing, 2-9
AllocMemoryInit, 16-32
AlphaColorEnabled, 3-27 to 3-28
alt requests, 5-4
amplifying voice messages, 14-11
analog crosspoint switch array, 14-9
Async.lib
  common-code module, 16-25
  context control block, 16-27
  contexts, 16-26, 16-28
  managing contexts, 16-25
  procedures, 16-10, 16-25
  requesting a service, 16-10
Asynchronous System Service, 16-1
  context stack, 16-15
  debugging, 16-22
  debugging statistics, 16-24
  file availability, 16-33
  linking, 16-34
  LogAsync, 16-22
  procedural interface, 16-11
  program examples, 16-36
  program flow, 16-7
  request blocks, 16-14
  requests, 16-10 to 16-11
  source modules, 16-9
  synchronous and, 16-3
  terminology, 16-1
  variable length parameter list, 16-12
  writing, 16-6
Asynchronous Terminal Emulator (ATE), 14-2
AsyncRequest, 16-14
AsyncRequestDirect, 16-14
attribute byte, 3-24, 4-3
attribute map, 4-23
audio features,
  of CD-ROM, 17-24
Audio Pause, 17-25
audio play, 17-24
  example of, 17-25
Audio Q-Channel Info function, of CD-ROM Service, 17-24
Audio Resume, 17-25
Audio Status, 17-25
automatic data segment, 10-18 (See also DGroup)
automating DLL building, 10-22 to 10-26
awk, 1-1
a-characters,
  CD-ROM files, 17-59

B

background color, 3-2, 4-3
backslash, in CD-ROM file specification, 17-8
batch, 6-6, 6-23
Batch.run, 6-1
baud rate, 9-4
blank space, on tape, 18-7
block size, 18-17
blocks, increasing size of, 18-17
buffers
  determining size of, 18-8
  recovery order, 18-12, 18-19
  specifying size of, 18-16
BuildAsyncRequest, 16-11
BuildAsyncRequestDirect, 16-12

C

call gate, 2-12
call progress tone detection (CPTD), 14-9, 14-18
CD-ROM disc
  directory list buffer, 17-9
  flat file structure, 17-6
  hierarchical file structure, 17-6
  primary volume descriptor, 17-4
CdAbsoluteRead, 17-4
CdAudioCtl, 17-3, 17-24
  example, 17-25
CdClose, 17-3
CdControl, 17-3
CdDirectoryList, 17-2, 17-9 to 17-10
  example of, 17-11
CdGetDirEntry, 17-2, 17-19, 17-49
CdGetVolumeInfo, 17-2, 17-4
CdOpen, 17-3
CdRead, 17-3
CdSearchClose, 17-2, 17-13
CdSearchFirst, 17-2, 17-13
CdSearchNext, 17-2, 17-13
CdServiceControl, 17-4
CdVerifyPath, 17-3
CdVersion, 17-3
CdVersionRequest, 17-13
CD-ROM
  character sets, 17-58
  example of file search, 17-13 to 17-14
  file, copying to disc, 17-19
  file formats, 17-33
  High Sierra directory record format, 7-54
  High Sierra primary volume descriptor, 7-39
  ISO directory record format, 17-49
  ISO primary volume descriptor, 17-33
  searching for files, 17-1
CD-ROM disc
  file structures used, 17-6
  specifying locations on, 17-24
CD-ROM Service
  audio features, 17-24
  file formats, 17-4
  function of, 17-1
  installation,17-1
  operations, 17-2
  requirements for, 17-1
ChangeCommLineBaudRate, 9-4
character
  attribute map, 4-23
character attribute byte, 3-24
character attributes, 4-17
character cell, 3-2, 4-2
character code, 4-3
character color, 3-25

character map, 3-3, 4-22
  accessing, 4-40
character sets
  for CD-ROM, 17-58
CheckContextStack, 16-15
child partition termination status, 5-6
circular queue, creating, 10-7 to 10-8
CleanQueue, 12-8
cleanup procedures (See DLL termination procedures)
Clients.mak, 10-55 to 10-57
client-server model, 1-1
clock source, 9-5
cluster network, 1-1
cluster server, 18-1
CODEC (encoder/decoder), 14-3, 14-6, 14-10, 14-23, 14-27
color intensity
  three-palette format, 3-10
color intensity, 3-7, 3-9
color programming, 3-1
  and graphics, 3-5
  character attribute byte, 3-24
  color priority, 3-28
  graphics colors, 3-27
  palette control structure, 3-11
  single-palette format, 3-7
  three-palette color format, 3-9
command file editor, 6-6, 6-14
command file, for installation, 6-6, 6-14
CommLine interface, 9-1
  baud Rate, 9-4
  clock source, 9-5
  extensions, 9-1
  reading signal status, 9-4
  serial controller differences, 9-3
  setting signal status, 9-5
  using DMA with, 9-5
communication controller, 9-3
communications DMA, 9-5
  and DTR signal, 9-5
  and WriteCommLineStatus, 9-5
  external/Status interrupt, 9-7
  getting status, 9-8
  initializing, 9-6
  receive special interrupt, 9-8
  receiving data, 9-7
  transmitting data, 9-7
communications line configuration block, 9-2
  fDMA field, 9-6
  fV35Mode field, 9-13
  fX21 field, 9-12
communications line return area, 9-2
  dMAAvailable field, 9-6
  fV35Avail field, 9-13
  ioX21 field, 9-11
compiler options, CTOS Microsoft C 10-1
compiling and building DLLs, 10-19 to 10-26
compiling DLL clients, 10-31
compiling DLLs, 10-23
compressing pauses
  in voice files, 14-10
Config.sys, 3-14
  parameters, 1-17
configuration
  spooler, 13-1
ConfigureSpooler, 13-2
connection handle, 8-20
context control block, 16-27
Context Manager, 5-1, 5-6, 5-8, 6-15, 6-24, 14-3

contexts, 16-26
  control block, 16-27
  creating, 16-28
  terminating, 16-29
  uses, 16-28
control file, for installation, 6-6 to 6-7
converting CTOS VGA, 4-1
  concepts and terminology, 4-2
  mapping character attributes, 4-23
  VAM 4.1.5, 4-1
copying a CD-ROM file to disc, example of, 17-19
CPTD configuration file, 14-19
Create Message File command, 6-13
CreateContext, 16-26
creating DLL import libraries, 10-24
creating DLL object modules, 10-24
creating dynamic link libraries, 10-1 to 10-57
creating partitions, 5-2
CTOS, 1-1
  calling convention, 1-18
  development tools, 1-2
  model of segmentation used, 1-17
  naming conventions, 10-6
  protection model used, 2-13
  system calls, 1-5
  system debugger, 1-19
  system software, 1-2
  use of call gates, 2-12
  use of GDT-based segments, 2-8
CTOS character attributes, 4-20
CTOS III, 10-1
CTOS naming conventions, 10-6
CTOS semaphore functions, 10-14
CTOS VGA
  converting to standard VGA, 4-1
  VAM 4.1.5, 4-1
  VAM default colors, 4-25
  default colors, 4-21
CTOS.lib version consistency, 1-20
CTOS/XE, 8-1
  exchanges and user numbers, 8-2
  standard connection handle, 8-21
CtosLib.dll, 10-2
CtosLibimp.lib, 10-3
cursor
  graphics, 11-14
  movement, 11-15
  shape, 11-12
cyclic redundancy check (CRC), 9-7
c-characters,
  CD-ROM files, 17-60

D

d1-characters,
  CD-ROM files, 17-60
data
  global, 10-6
  instance 10-6
  nonshared (See instance data)
  reading and writing, 14-24
  shared (See global data)
data and voice
  separate lines for, 14-15
data blocks
  on tape, 18-6
data call
  accepting, 14-23
  converting a voice call to, 14-23
  example, 14-22
  originating, 14-24
  starting, 14-23
  terminating, 14-24

data control structure, 14-23
data management, 4-2, 14-2
data statement, 10-18
data storage characteristics
  of sequential access devices, 18-4
data terminal ready (DTR), 9-5
data types, 1-9
  defining, 1-10
data, global 10-6
DDS devices, 18-18
  recording data on, 18-6
debugger, 1-19
default attribute mapping tables, 4-29
  monochrome, 4-30 to 4-31
  with alphanumeric color, 4-32 to 4-39
  without alphnumeric color, 4-30
default character attribute mapping
  standard VGA, 4-24
default colors
  CTOS VGA, 4-21
  standard VGA, 4-19
default system font, 4-10
defining DLL default data types, 10-18
defining DLL instance data, 10-18
DeinstallQueueManager, 12-8
dependents, 10-23
descriptor table, 2-4
development library consistency, 1-20
device routing, 8-4
devices, supported by Sequential Access Service, 18-1
DGroup, 10-18
dialer, 14-28
dialing telephone numbers, 14-16
digital data storage (DDS), 18-1
digitizing voice, 4-3
directory list buffer
  for CD-ROM disc, 17-9
directory list
  for CD-ROM disc, 17-9
disk activity, 5-5, 15-1
disk requirements
  for voice files, 14-12
disk seeks, 14-12
display resolution, 4-15
distributed computing, 1-1
DLL
  source files, 10-5
  client, 10-4, 10-27
  initialization procedures, 10-11 to 10-12
  initialization types, 10-11
  instance data, defining 10-18
  termination procedures, 10-12 to 10-16
DLL.mak, 10-47 to 10-48
DLL_Term.obj, 10-3
DMA for communications (See communications DMA)
DTMF
  encoder, 14-3
  generator and receiver, 14-3, 14-7
  generating tones, 14-17 to 14-18
dynamic link libraries
  creating, 10-1 to 10-57
dynamic linking
  loadtime 10-2
  runtime 10-2
d-characters,
  CD-ROM files, 17-58

E

early warning (EW), of sequential access devices, 18-4
edf files, 1-20
editor, 1-6
end of data frame (EOF), 9-8
End of Medium condition, 18-9
enhanced video, 3-2
EnterDebuggerOnFault, 1-17
Environment Service, 10-3, 10-27
erase gaps, on tape, 18-7
error checking, 10-5
EstablishQueueServer, 12-8
exception, 2-11
executive, 1-1, 3-4, 6-6
exit procedures (See DLL termination procedures)
ExitListSet, 10-12 to 10-13
export procedures, 10-10, 10-11
external/Status interrupt, 9-7, 9-11

F

fault, 2-11
fBackgroundColor, 3-13, 3-16, 3-28
fDataBufRecoverable, 18-8
FIFO, 10-4
file formats
  for CD-ROM, 17-4, 17-33
file handle, 8-20
file suffix conventions,
file system activity, 15-1
filemarks, 18-20
  on tape, 18-7
fixed-length records, 18-16
flat file structure
  CD-ROM disc, 17-6
floppy installation, 6-2
  naming files, 6-16
font, 4-3
foreground color, 3-2, 4-3
fSuppressDefaultOnOpen, 18-19

G

gaps, on tape, 18-7
gate descriptor, 2-12
general protection fault, 2-11
GetCommLineDMAStatus, 9-8
GetQmStatus, 12-8
GetWsUserName operation, 8-3
global data variables, 10-17
global descriptor table, 2-8
global DLL initialization, 10-11
graphics and color, 3-5
GraphicsColorEnabled, 3-29
GraphicsEnabled, 3-29
gray-scale monitors, 3-23

H

half-bit shift, 4-12
half-inch reel-to-reel devices, 18-1, 18-18
  recording data on, 18-6
handle, 8-20, 8-21
header files, 1-20
heap memory, 16-15
  allocating, 16-16
  conserving, 16-17
  deallocating, 16-16
  debugging, , 16-32
  initializing, 16-32
  managing, 16-30
  using, 16-29

HeapAlloc, 16-16, 16-31
HeapFree, 16-16, 16-31
HeapInit, 16-30
helical scan recording, 18-6
hierarchical file structure
  for CD-ROM disc, 17-6
High Sierra directory record format, 7-54
High Sierra primary volume descriptor
  for CD-ROM, 7-39
High Sierra standard for CD-ROM, 17-1
hold
  placing a telephone line on, 14-16
horizontal scan line, 4-3

I

InitCommLine, 9-2
  and DMA, 9-6
  selecting extended features, 9-2
  V.35 protocol support, 9-13
  X.21 protocol support, 9-12
initialization, DLL, 10-11
Install Sequential Access Service command, 18-1
installation database, 6-4
Installation Manager
  file lists created by, 6-27
  verify feature, 6-10
installation media, 6-2
  organizing, 6-20
installation scripts
  examples, 6-34
  file, 6-6, 6-13
  tips for creating, 6-30
installation variables, 6-4, 6-23
installation, restarting, 6-28
instance data, 10-6
instance DLL initialization, 10-11
Intel documentation titles, 2-1
intercharacter space, 4-4
internationalized call progress tone detection, 14-18
  example of, 14-21
interrupt service routine
  and DMA, 9-5
  and External/Status interrupt, 9-7
  and receive special interrupt, 9-8
inter-CPU communication, 8-3
ISO directory record format, for CD-ROM, 17-49
ISO primary volume descriptor, for CD-ROM, 17-33
ISO-9660 standard, for CD-ROM, 17-1

J

JCL files, 6-1, 6-13
  examples, 6-34

K

kernel, 1-3

L

LIBRARY statement, 10-11
library version consistency, 1-20
Link command, 1-7
Link V6 command, 1-6, 17-10
linker configuration file, 10-3, 10-27 (See also *LinkerConfig.sys*)
linker, 1-6
LinkerConfig.sys, 10-3, 10-26, 10-27, 10-33, 10-57
linking DLL clients, 10-32 to 10-33
linking DLLs, 10-25
list file, 1-7
LLibCDll.lib, 10-3
LoadInteractiveTask operation, 5-6
loadtime linking, 10-2
local descriptor table, 2-7
local routing, 8-5, 8-18
locking semaphores, 10-15
log file, 7-1
  and volume home block, 7-3, 7-6
  chronological order, 7-7
  for installation, 6-10
  format of, 7-2
  reading, 7-6
  record header and trailer, 7-2
  wraparound, 7-5 to 7-6
  writing records to, 7-2
  written by file system, 7-4
LogAsync, 16-22
logging session
  closing, 15-10
  for Performance Statistics Service, 15-1, 15-8
  opening, 15-8
  reading a log, 15-9
logical address, 2-2
LogMsgIn, 16-22, 16-32
LogRequest, 16-22, 16-32
LogRespond, 16-22, 16-32

M

make, 1-1
makefiles, using 10-22 to 10-26
map file, 1-8
mapping character attributes, 4-23
mapping CTOS attributes, 4-41
MarkKeyedQueueEntry, 12-8
MarkNextQueueEntry, 12-8
MC0_Dll.obj, 10-3
mediating medium-model procedures, 10-10
memory addressing, 2-2
Merge Command Files command, 6-14
message file, for installation, 6-6, 6-13
minute-second-frame (MSF) format, on CD-ROM disc, 17-24
mixing programming languages, 1-18
models of segmentation, 1-17
modem, 14-1 to 4-2, 14-5, 14-9, 14-28
  asynchronous use of, 14-24
Module Definition command, 10-24
module definition file, 10-6, 10-11, 10-18
monochrome graphics, 3-24
motion rectangle, 11-7

Mouse Services, 6-7, 6-14, 11-1
  important concecpts, 11-3
  mouse buttons, 11-4
  procedures, 11-2
  programming examples, 11-3
  queries, 11-7
  screen coordinates, 11-4, 11-5
  setup, 11-6
  troubleshooting, 11-16
multiple voice messages
  in one file, 14-14

N

naming conventions, 1-9, 10-6
naming floppy installation files, 6-16
naming tape installation files, 6-18
nationalization, 6-13, 6-29
network, 14-2
nine-bit font capabilities, 4-6
NMake, 10-3, 10-27
nonshared data (See DLL instance data)
NotifyCM request, 5-6
NULL pointer, 2-11

O

object file, 1-7
object module library version consistency, 1-20
object module library, 11-1
object module procedure, 1-3
obtaining semaphores, 10-15
offhook, 14-16
operating system calls, 1-3
operations for updating the screen, 4-42
operator, 14-2
overriding DLL defaults, 10-18

P

package, 6-4
paging, 2-6
palette
  alphanumeric vs. graphics, priority, 3-28
palette control structure, 3-11
  ProgramColorMapper
palettes, 3-6
  sample, 3-17
paragraph, 2-3
parallel recording, 18-6
parity, 14-5
partition management, 5-1
  Action+Finish and swapping, 5-7
  and child termination, 5-6
  sample program, 5-10
  termination procedure, 5-8
  use of termination requests, 5-7
partition type, 5-3
partitions
  consequences of unsuccessful task load, 5-6
  creating, 5-2
  deallocating, 5-3
  deallocating, 5-9
  initializing, 5-4
  loading a task into, 5-6
password
  spooler, 13-2
pause compression
  advantages and disadvantages of, 14-11
pause compression, 14-10
PBX systems, 14-7, 14-16
PDLoadCursor, 11-14
PDReadIconFile, 11-14
PDSetCursorDisplay, 11-14
PDSetCursorPos, 11-7

PDSetCursorType, 11-6
PDSetTracking, 11-7
PDSetVirtualCoordinates, 11-6
PDTranslateNSCtoVC, 11-5
PDTranslateVCtoNSC, 11-5
Pentium, 2-3 to 2-4
Performance Statistics Service
  example of, 15-10
  function of, 15-1
performance statistics structure, 15-4, 15-17
pixel, 3-2
playback, of compressed voice files, 14-11
PLog utility, 7-1, 7-3
  record-processing algorithm, 7-10
porting to protected mode, 8-1
preparing DLL clients, 10-29 to 10-30
preparing DLLs, overview, 10-20 to 10-21
primary volume descriptor
  for CD-ROM disc, 17-4
private installation, 6-3
procedure naming, 1-14
procedures
  DLL initialization, 10-12
  DLL termination, 10-12
  ExitListSet, 10-12 to 10-13
  export, 10-11, 10-16
  setting priorities of, 10-13
processor activity, 15-1
ProgramColorMapper, 3-1
  and monochrome graphics, 3-24
  control structure, 3-6
  functions performed by, 3-5
  single-palette format, 3-7
  three-palette format, 3-9
programming languages, 1-2
protected mode
  address calculation in, 2-4
  descriptor tables, 2-7
  exceptions and faults, 2-11
  features of, 2-2
  introduction to, 2-1
protected mode run file, 1-6
PsCloseSession, 15-2, 15-10
PsGetCounters, 15-2, 15-3, 15-6
PsOpenLogSession, 15-2
PsOpenStatSession, 15-3, 15-4
PsReadLog, 15-2, 15-9
PsResetCounters, 15-2
public installation, 6-3, 6-10, 6-26

Q

QIC tape device, 18-16
  recording data on, 18-5
quarter-inch cartridge (QIC) tape, 18-1
QueryVideo, 3-14, 3-23
queue entries, 12-4
  adding, 12-6
  calculating size, 12-13
  deleting, 12-7
  description, 12-12
  format, 12-13
  handles, 12-18
  header, 12-27
  keys, 12-19
  marking, 12-4
  processing order, 12-12
  processing, 12-5
  queue management, 13-5
  reading, 12-7

referencing, 12-18
removing, 12-5
rescheduling, 12-5
rewriting, 12-5
spooler, 13-5
status blocks, 12-19
unmarking, 12-5
Queue Manager, 12-1
client operations, 12-6
configuration, 12-10
data structures, 12-24
deinstallation, 12-2
installation, 12-2
network nodes, 12-22
operations, 12-3
run files, 12-2
sequence, 12-21
summary of operations, 12-7
queue server, 12-4
queue, 10-4
Queue.c, 10-35 to 10-46
Queue.def, 10-6, 10-18, 10-46
Queue.h, 10-49
queues
defining dynamically, 12-16
defining, 12-14
examples, 12-14
file header, 12-11, 12-24
format, 12-11
handles, 12-18
index file, 12-14
names, 12-17
referencing, 12-17
status block, 12-29
Q-channel, 17-24

R

ReadCommLineStatus, 9-4
ReadKeyedQueueEntry, 12-8
ReadNextQueueEntry, 12-8
ReadQueue.c, 10-50
real mode address calculation, 2-3
receive special interrupt, 9-8
ReceiveCommLineDma, 9-7
record size, 18-17
record/playback
typical sequence for, 14-14
recording data
on half-inch tape devices, 18-6
on QIC tape devices, 18-5
recording densities, 18-4, 18-18
recording rates
for voice, 14-10
recording voice, 14-10
records, Sequential Access Service, 18-16
recovering buffer data, 18-9, 18-13
Red Book standard, 17-24
remote processor memory, 8-3
remote routing, 8-4, 8-15
RemoveKeyedQueueEntry, 12-9
RemoveMarkedQueueEntry, 12-9
RemoveQueue, 12-9
request blocks, 16-14
request routing, 8-3
across the cluster, 1-2
inter-board routing directives, 8-4
request, 1-3

requests
aborting, 16-17
handling, 16-19
swapping, 16-18
terminating, 16-17
RescheduleMarkedQueueEntry, 12-9
ResetVideoGraphics, 3-14, 3-23
residual data, 18-9 to 18-10
Resource Librarian command, 10-26, 10-34
restarting an installation, 6-28
restartLabel, 6-28
Restore command, 17-10
ResumeContext, 16-27
RewriteMarkedQueueEntry, 12-9
routing by device specification, 8-6
run command, 1-8
run file mode, 1-6, 2-8
run file, 1-7
runtime linking, 10-2

S

SCSI devices, 18-19
SCSI interfaces, 18-1
Searching
for CD-ROM files, 17-13
segment address, 2-4
segment descriptor format, 2-9
segmented addressing, 2-2
in protected mode, 2-5
segments statement, 10-6, 10-18
selector, 2-4
format of, 2-7
semaphore functions, CTOS 10-14
SemClear, 10-16
SemLock, 10-14, 10-16
SemOpen, 10-14
Separators,
CD-ROM files, 17-60
SeqAccessCheckpoint, 18-3, 18-14
SeqAccessClose, 18-2, 18-12
SeqAccessControl, 18-2, 18-14
SeqAccessDiscardBufferData, 18-3, 18-10
SeqAccessModeQuery, 18-3, 18-8
SeqAccessModeSet, 18-3, 18-12
SeqAccessOpen, 18-2
SeqAccessRead, 18-2
SeqAccessRecoverBufferData, 18-3
SeqAccessStatus, 18-2
SeqAccessVersion, 18-3
SeqAccessWrite, 18-2, 18-9
sequential access devices
data storage characteristics of, 18-4
model of, 18-4
Sequential Access Service
data buffering, 18-8
determining size of service buffers, 18-8, 18-9
devices supported by, 18-1
function of, 18-1
installing multiple instances, 18-1
placement of in cluster, 18-1
programming considerations, 18-17
records, 18-16
recovering buffer data, 18-9
SequentialAccessControl, 18-7
serial controller, 9-3
and DMA, 9-6
CTS signal and X.21, 9-10
external/Status interrupt, 9-7
initializing for X.21, 9-12
receive special interrupt, 9-8

serpentine recording, 18-5
server installation, 6-2
shared data (See global data)
shared resource processor (SRP), 8-3, 18-1
  and GetWsUserName, 8-3
  device specification, 8-6
  exchanges on, 8-2
  inter-CPU communication on, 8-3
  multi-instance system services, 8-6
  porting real-mode applications, 8-1
  programming guidelines, 8-2
  request routing, 8-3
  sample *request.txt* file for, 8-7, 8-15
  special handle types, 8-21
  use of handles, 8-20
  user numbers on, 8-2
simple guidelines for
  creating makefiles, 10-22
single-palette color format, 3-7
  advantages of, 3-9
source code files, 1-6
spawn, 5-1
spooler
  configuration, 13-1
  control queue, 13-6, 13-14
  data structures, 13-8
  escape sequences, 13-5
  file requirements, 13-4
  introduction, 13-1
  operations, 13-2
  password, sending, 13-2
  pre-GPS, 13-3
  program examples, 13-7
  programmer's notes, 13-3
  queue management, 13-5
  queue scheduling, 13-6
  scheduling queue, 13-8
  status queue, 13-6, 13-11
SpoolerPassword, 13-2
standard connection handle, 8-21
standard VGA
  character attributes, 4-26 to 4-29
  default character attribute mapping, 4-24
  default colors, 4-19
statistics ID block, 15-3, 15-4
statistics session,
  closing, 15-8
  for Performance Statistics Service, 15-1
  opening, 15-4
subpackages, 6-3
SwapContextUser, 16-21
SwapInContext, 5-2
symbol file, 1-8
synchronous data communication, 9-1
  baud Rate, 9-4
  clock source, 9-5
  extensions to traditional interface, 9-1
  reading signal status, 9-4
  selecting extended features, 9-2
  serial controller differences, 9-3
  setting signal status, 9-5
  using DMA, 9-5
  V.35 protocol support, 9-13
  X.21 protocol support, 9-9
synchronous system service
  and asynchronous, 16-3
  model program flow, 16-5
  procedural interface, 16-10
syntax, makefiles 10-22 to 10-26
system configuration tips, 1-19

system log file, 7-1
  and volume home block, 7-3, 7-6
  chronological order, 7-7
  format of, 7-2
  reading, 7-6
  record header and trailer, 7-2
  wraparound, 7-5 to 7-6
  writing records to, 7-2
  written by file system, 7-4
system service, 8-6
System.dll, 10-2
Systemimp.lib, 10-3
system-common procedure, 1-3
  and the GDT, 2-8

T

tape
  logical elements within, 18-7
tape installation, 6-2
  naming files, 6-18
  organizing files, 6-23
tape, general layout of, 18-5
targets, 10-23
Task Management Service, 10-3, 10-27
telephone lines, 14-7
telephone management, 14-2
telephone Service, 4-1
  data call example, 14-69
  dialing example, 14-30
  voice memory playback example, 14-57
  voice response system example, 14-36
telephone service configuration block, 14-4
TerminateContextUser, 16-19
TerminateQueueServer, 12-9
termination requests, 5-7, 5-18
three-palette color format, 3-9
track number, on CD-ROM disc, 17-24
track, definition of, 18-5
TransmitCommLineDma, 9-7
transport speed, 18-18
troubleshooting, 1-20
TsConnect operation, 14-3, 14-3
TsDataChangeParams operation, 14-5
TsDataCheckpoint operation, 14-5
TsDataClose operation, 14-23
TsDataCloseLine operation, 14-5
TsDataOpenLine operation, 14-5, 14-23, 14-24
TsDataRead operation, 14-5, 14-23, 14-24
TsDataRetrieveParams operation, 14-5
TsDataUnacceptCall operation, 14-5
TsDataWrite operation, 14-5, 14-23, 14-24
Telephone Status command
  function keys, 14-29
  screen, 14-26
Telephone Status command, 14-1 to 14-2, 14-25
telephone status monitor program, 14-2
telephone status structure, 4-4
telephone unit
  parts of, 4-9, 14-7, 14-28
  versus telephone line, 14-16

TerminateContext, 16-28
TsDeinstall operation, 14-3
TsDial operation, 14-3, 14-17
TsDoFunction operation, 14-3, 14-16
TsGetStatus operation, 14-4
TsHold operation, 14-4, 14-16
TsLoadCallProgressTones operation, 14-4
TsOffHook operation, 14-4, 14-16
TsOnHook operation, 14-4, 14-16
TsQueryConfigParams operation, 14-4
TsReadTouchTone operation, 14-4
TsRing operation, 14-4
TsSetConfigParams operation, 14-4
TsVersion operation, 14-4
TsVoiceConnect operation, 14-3
TsVoicePlaybackFromFile operation, 14-3, 14-15
TsVoiceRecordToFile operation, 14-3
TsVoiceStop operation, 14-3

U

UnmarkQueueEntry, 12-9
user configuration file, 6-15
user number, 8-2
using DLLs, 10-27
using macros, 10-22
using makefiles, 10-22 to 10-26

V

V.35 protocol support hardware, 9-13
variable length parameter list, 16-12
variable names, 1-10
  examples, 1-13
  prefixes, 1-10
  roots, 1-12
  suffixes, 1-13
variable-length records, 18-16, 18-18, 18-25
VGA Character Attribute Mapper, 4-23
VGA, 3-2, 3-14
Video Graphics Array (See VGA)
video resource file, 4-23
video, during installation, 6-15
virtual address, 2-5
virtual screen coordinates, 11-4
voice amplifier, 14-6
voice and data
  separate lines for, 14-15
voice control structure, 14-3, 14-10, 14-14
voice file
  structure of a, 14-13
voice file header, 14-13
voice file record, 14-13
voice playback
  from memory, 14-15

voice processor module, 14-1
  connections, 4-10
  data features of, 14-9
  voice features of, 14-6
voice recording, 14-10
voice response system, 14-8
Voice/Data Services
  debugging using Telephone Status command, 14-25
Voice/Data Services
  definition of, 14-1
  functional operation groups, 14-2
  hardware features used by, 14-6
volume home block
  and log file, 7-3, 7-6

W

work area
  for the Telephone Service, 14-12
WriteCommLineStatus, 9-5
WriteQueue.c, 10-51 to 10-54
writing filemarks on tape, 18-7

X

X-Bus workstations, 14-1
X.21 protocol
  general description, 9-9
  signal lines used, 9-9
  special signaling bit patterns, 9-10
X.21 protocol support, 9-9
  and external/status interrupt, 9-11
  hardware features, 9-10
  initializing communications with, 9-12
  use of CTS signal, 9-10
X.21 protocol support hardware
  drivers-only mode, 9-12
  enabling and disabling, 9-12
  features of, 9-10
XE-530, 8-1
XmitCommLineDMA, 9-7

Y

yacc, 1-1

45867900-000